Coccinelle

User's manual

release 0.1.8

Julia Lawall and Yoann Padioleau
(with contributions from Rene Rydhof Hansen, Nicolas Palix, Henrik Stuart)

October 21, 2009

Contents

Foreword

This manual documents the release 0.1.8 of Coccinelle. It is organized as follows:

- Part I is an introduction to Coccinelle
- Part II is the reference description of Coccinelle, its language and command line tool.

Conventions

Copyright

Availability

Coccinelle can be freely downloaded from `http://coccinelle.lip6.fr/`. This website contains also additional information.

Part I

User Manual

Chapter 1

Introduction

Coccinelle is a tool to help automate repetitive source-to-source style-preserving program transformations on C source code, like for instance to perform some refactorings. Coccinelle is presented as a command line tool called `spatch` that takes as input the name of a file containing the specification of a program transformation, called a *semantic patch*, and a set of C files, and then performs the transformation on all those C files.

To make it easy to express those transformations, Coccinelle proposes a WYSISWYG approach where the C programmer can leverage the things he already knows: the C syntax and the patch syntax. Indeed, with Coccinelle transformations are written in specific language called SmPL, for Semantic Patch Language, which as the name suggests is very close to the syntax of a patch, but which does not work at a line level, than traditional patches do. but a more high level, or semantic level.

Here is an example of a simple program transformation. To replace every calls to `foo` of any expression x to a call to `bar`, create a semantic patch file `ex1.cocci` (semantic patches usually ends with the `.cocci` filename extension) containing:

```
@@ expression x; @@

- foo(x)
+ bar(x)
```

Then to "apply" the specified program transformation to a set of C files, simply do:

```
$ spatch -sp_file ex1.cocci *.c
```

Chapter 2

Installing Coccinelle

2.1 Requirements

2.2 Getting Coccinelle

2.3 Compiling Coccinelle

2.4 Running Coccinelle

Chapter 3

Tutorial

Chapter 4

Examples

4.1 Examples

This section presents a range of examples. Each example is presented along with some C code to which it is applied. The description explains the rules and the matching process.

4.1.1 Function renaming

One of the primary goals of Coccinelle is to perform software evolution. For instance, Coccinelle could be used to perform function renaming. In the following example, every occurrence of a call to the function `foo` is replaced by a call to the function `bar`.

Before

```c
#DEFINE TEST "foo"

printf("foo");

int main(int i) {
//Test
  int k = foo();

  if(1) {
    foo();
  } else {
    foo();
  }

  foo();
}
```

Semantic patch

```
@@

@@


- foo()
+ bar()
```

After

```c
#DEFINE TEST "foo"

printf("foo");

int main(int i) {
//Test
  int k = bar();

  if(1) {
    bar();
  } else {
    bar();
  }

  bar();
}
```

4.1.2 Removing a function argument

Another important kind of evolution is the introduction or deletion of a function argument. In the following example, the rule `rule1` looks for definitions of functions having return type `irqreturn_t` and two parameters. A second *anonymous* rule then looks for calls to the previously matched functions that have three arguments. The third argument is then removed to correspond to the new function prototype.

```
@ rule1 @
identifier fn;
identifier irq, dev_id;
typedef irqreturn_t;
@@

static irqreturn_t fn (int irq, void *dev_id)
{
   ...
}

@@
identifier rule1.fn;
expression E1, E2, E3;
@@

 fn(E1, E2
-   ,E3
    )
```

`drivers/atm/firestream.c` at line 1653 before transformation

```
static void fs_poll (unsigned long data)
{
        struct fs_dev *dev = (struct fs_dev *) data;

        fs_irq (0, dev, NULL);
        dev->timer.expires = jiffies + FS_POLL_FREQ;
        add_timer (&dev->timer);
}
```

`drivers/atm/firestream.c` at line 1653 after transformation

```
static void fs_poll (unsigned long data)
{
        struct fs_dev *dev = (struct fs_dev *) data;

        fs_irq (0, dev);
        dev->timer.expires = jiffies + FS_POLL_FREQ;
        add_timer (&dev->timer);
}
```

4.1.3 Introduction of a macro

To avoid code duplication or error prone code, the kernel provides macros such as BUG_ON, DIV_ROUND_UP and FIELD_SIZE. In these cases, the semantic patches look for the old code pattern and replace it by the new code.

A semantic patch to introduce uses of the DIV_ROUND_UP macro looks for the corresponding expression, *i.e.*, $(n + d - 1)/d$. When some code is matched, the metavariables n and d are bound to their corresponding expressions. Finally, Coccinelle rewrites the code with the DIV_ROUND_UP macro using the values bound to n and d, as illustrated in the patch that follows.

Semantic patch to introduce uses of the DIV_ROUND_UP macro

```
@ haskernel @
@@

#include <linux/kernel.h>

@ depends on haskernel @
expression n,d;
@@

(
- (((n) + (d)) - 1) / (d))
+ DIV_ROUND_UP(n,d)
|
- (((n) + ((d) - 1)) / (d))
+ DIV_ROUND_UP(n,d)
)
```

Example of a generated patch hunk

```
--- a/drivers/atm/horizon.c
+++ b/drivers/atm/horizon.c
@@ -698,7 +698,7 @@ got_it:
                if (bits)
                        *bits = (div<<CLOCK_SELECT_SHIFT) | (pre-1);
                if (actual) {
-                       *actual = (br + (pre<<div) - 1) / (pre<<div);
+                       *actual = DIV_ROUND_UP(br, pre<<div);
                        PRINTD (DBG_QOS, "actual␣rate:␣%u", *actual);
                }
                return 0;
```

The BUG_ON macro makes a assertion about the value of an expression. However, because some parts of the kernel define BUG_ON to be the empty statement when debugging is not wanted, care must be taken when the asserted expression may have some side-effects, as is the case of a function call. Thus, we create a rule introducing BUG_ON only in the case when the asserted expression does not perform a function call.

On particular piece of code that has the form of a function call is a use of unlikely, which informs the compiler that a particular expression is unlikely to be true. In this case, because unlikely does not perform any side effects, it is safe to use BUG_ON. The second rule takes care of this case. It furthermore disables the isomorphism that allows a call to unlikely be replaced with its argument, as then the second rule would be the same as the first one.

```
@@
expression E,f;
@@

(
  if (<+... f(...) ...+>) { BUG(); }
|
- if (E) { BUG(); }
+ BUG_ON(E);
)

@ disable unlikely @
expression E,f;
@@

(
  if (<+... f(...) ...+>) { BUG(); }
|
- if (unlikely(E)) { BUG(); }
+ BUG_ON(E);
)
```

For instance, using the semantic patch above, Coccinelle generates patches like the following one.

```
--- a/fs/ext3/balloc.c
+++ b/fs/ext3/balloc.c
@@ -232,8 +232,7 @@ restart:
                        prev = rsv;
                }
                printk("Window␣map␣complete.\n");
-               if (bad)
-                       BUG();
+               BUG_ON(bad);
 }
 #define rsv_window_dump(root, verbose) \
                __rsv_window_dump((root), (verbose), __FUNCTION__)
```

4.1.4 Look for **NULL** dereference

This SmPL match looks for `NULL` dereferences. Once an expression has been compared to `NULL`, a dereference to this expression is prohibited unless the pointer variable is reassigned.

Original

```
foo = kmalloc(1024);
if (!foo) {
  printk ("Error␣%s", foo->here);
  return;
}
foo->ok = 1;
return;
```

Semantic match

```
@@
expression E, E1;
identifier f;
statement S1,S2,S3;
@@

* if (E == NULL)
{
  ... when != if (E == NULL) S1 else S2
      when != E = E1
* E->f
  ... when any
  return ...;
}
else S3
```

Matched lines

```
foo = kmalloc(1024);
if (!foo) {
  printk ("Error %s", foo->here);
  return;
}
foo->ok = 1;
return;
```

4.1.5 Reference counter: the of_xxx API

Coccinelle can embed Python code. Python code is used inside special SmPL rule annotated with `script:python`. Python rules inherit metavariables, such as identifier or token positions, from other SmPL rules. The inherited metavariables can then be manipulated by Python code.

The following semantic match looks for a call to the `of_find_node_by_name` function. This call increments a counter which must be decremented to release the resource. Then, when there is no call to `of_node_put`, no new assignment to the `device_node` variable `n` and a `return` statement is reached, a bug is detected and the position `p1` and `p2` are initialized. As the Python only depends on the positions `p1` and `p2`, it is evaluated. In the following case, some emacs Org mode data are produced. This example illustrates the various fields that can be accessed in the Python code from a position variable.

```
@ r exists @
local idexpression struct device_node *n;
position p1, p2;
statement S1,S2;
expression E,E1;
@@

(
if (!(n@p1 = of_find_node_by_name(...))) S1
|
n@p1 = of_find_node_by_name(...)
)
<... when != of_node_put(n)
    when != if (...) { <+... of_node_put(n) ...+> }
    when != true !n  || ...
    when != n = E
    when != E = n
if (!n || ...) S2
...>
(
  return <+...n...+>;
|
return@p2 ...;
|
n = E1
|
E1 = n
)

@ script:python @
p1 << r.p1;
p2 << r.p2;
@@

print "* TODO [[view:%s::face=ovl-face1::linb=%s::colb=%s::cole=%s][inc. counter:%s::%s]]" % (p1[0].file,p1[0].line,p1[0].column,p1[0].column_end,p1[0].file,p1[0].line)
print "[[view:%s::face=ovl-face2::linb=%s::colb=%s::cole=%s][return]]" % (p2[0].file,p2[0].line,p2[0].column,p2[0].column_end)
```

Lines 13 to 17 list a variety of constructs that should not appear between a call to `of_find_node_by_name` and a buggy return site. Examples are a call to `of_node_put` (line 13) and a transition into the then branch of a conditional testing whether `n` is `NULL` (line 15). Any number of conditionals testing whether `n` is `NULL` are allowed as indicated by the use of a nest `<... ...>` to describe the path between the call to `of_find_node_by_name`, the return and the conditional in the pattern on line 18.

The previously semantic match has been used to generate the following lines. They may be edited using the emacs Org mode to navigate in the code from a site to another.

```
* TODO [[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-face1::linb=236::colb=18::cole=20][inc. counter:/linux-next/arch/powerpc/platforms/pseries/setup.c::236]]
[[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-face2::linb=250::colb=3::cole=9][return]]
* TODO [[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-face1::linb=236::colb=18::cole=20][inc. counter:/linux-next/arch/powerpc/platforms/pseries/setup.c::236]]
[[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-face2::linb=245::colb=3::cole=9][return]]
```

Note : Coccinelle provides some predefined Python functions, *i.e.*, `cocci.print_main`, `cocci.print_sec` and `cocci.print_secs`. One could alternatively write the following SmPL rule instead of the previously presented one.

```
@ script:python @
p1 << r.p1;
p2 << r.p2;
@@

cocci.print_main(p1)
cocci.print_sec(p2,"return")
```

The function `cocci.print_secs` is used when there is several positions which are matched by a single position variable and that every matched position should be printed.

Any metavariable could be inherited in the Python code. However, accessible fields are not currently equally supported among them.

4.1.6 Filtering identifiers, declarers or iterators with regular expression

If you consider the following SmPL file which uses the regexp functionality to filter the identifiers that contain, begin or end by `foo`,

```
@anyid@
type t;
identifier id;
@@
t id () {...}

@script:python@
x << anyid.id;
@@
print "Identifier: %s" % x

@contains@
type t;
identifier foo ~= ".*foo";
@@
t foo () {...}

@script:python@
x << contains.foo;
@@
print "Contains foo: %s" % x

@endsby@
type t;
identifier foo ~= ".*foo$";
@@

t foo () {...}

@script:python@
x << endsby.foo;
@@
print "Ends by foo: %s" % x

@beginsby@
type t;
identifier foo ~= "^foo";
@@
t foo () {...}

@script:python@
x << beginsby.foo;
@@
print "Begins by foo: %s" % x
```

and the following C program, on the left, which defines the functions `foo`, `bar`, `foobar`, `barfoobar` and `barfoo`, you will get the result on the right.

```
int foo () { return 0; }
int bar () { return 0; }
int foobar () { return 0; }
int barfoobar () { return 0; }
int barfoo () { return 0; }
```

```
Identifier: foo
Identifier: bar
Identifier: foobar
Identifier: barfoobar
Identifier: barfoo
Contains foo: foo
Contains foo: foobar
Contains foo: barfoobar
Contains foo: barfoo
Ends by foo: foo
Ends by foo: barfoo
Begins by foo: foo
Begins by foo: foobar
```

4.2 Tips and Tricks

4.2.1 How to remove useless parentheses?

If you want to rewrite any access to a pointer value by a function call, you may use the following semantic patch.

```
- a = *b
+ a = readb(b)
```

However, if for some reason your code looks like `bar = *(foo)`, you will end up with `bar = readb((foo))` as the extra parentheses around `foo` are capture by the metavariable `b`.

In order to generate better output code, you can use the following semantic patch instead.

```
- a = *(b)
+ a = readb(b)
```

And rely on your standard.iso isomorphism file which should contain:

```
Expression
@ paren @
expression E;
@@

 (E) => E
```

Coccinelle will then consider `bar = *(foo)` as equivalent to `bar = *foo` (but not the other way around) and capture both. Finally, it will generate `bar = readb(foo)` as expected.

Chapter 5

Isomorphisms and `standard.iso`

Chapter 6

Parsing C, `cpp`, and `standard.h`

Chapter 7

Developing a Semantic Patch

Chapter 8

Advanced Features

Part II

Reference Manual

Chapter 9

SmPL grammar

This document presents the grammar of the SmPL language used by the Coccinelle tool. For the most part, the grammar is written using standard notation. In some rules, however, the left-hand side is in all uppercase letters. These are macros, which take one or more grammar rule right-hand-sides as arguments. The grammar also uses some unspecified nonterminals, such as id, const, etc. These refer to the sets suggested by the name, *i.e.*, id refers to the set of possible C-language identifiers, while const refers to the set of possible C-language constants. A HTML version of this documentation is available online at `http://coccinelle.lip6.fr/docs/main_grammar.html`.

9.1 Program

program	::=	*include_cocci** *changeset*+
include_cocci	::=	`using` *string*
	|	`using` *pathToIsoFile*
changeset	::=	*metavariables* *transformation*
	|	*script_metavariables* script_code

script_code is any code in the chosen scripting language. Parsing of the semantic patch does not check the validity of this code; any errors are first detected when the code is executed. Furthermore, `@` should not be use in this code. Spatch scans the script code for the next `@` and considers that to be the beginning of the next rule, even if `@` occurs within e.g., a string or a comment.

9.2 Metavariables for transformations

The *rulename* portion of the metavariable declaration can specify properties of a rule such as its name, the names of the rules that it depends on, the isomorphisms to be used in processing the rule, and whether quantification over paths should be universal or existential. The optional annotation `expression` indicates that the pattern is to be considered as matching an expression, and thus can be used to avoid some parsing problems.

The *metadecl* portion of the metavariable declaration defines various types of metavariables that will be used for matching in the transformation section.

```
metavariables       ::= @@ metadecl* @@
                      | @ rulename @ metadecl* @@
rulename            ::= id [extends id] [depends on dep] [iso] [disable-iso] [exists] [expression]
                      | script:language [depends on dep]
script_init_final   ::= initialize:language
                      | finalize:language
dep                 ::= pnrule
                      | dep && dep
                      | dep || dep
pnrule              ::= id
                      | !id
                      | ever id
                      | never id
                      | (dep)
iso                 ::= using string (, string)*
disable-iso         ::= disable COMMA_LIST(id)
exists              ::= exists
                      | forall
COMMA_LIST(elem)    ::= elem (, elem)*
```

The keyword `disable` is normally used with the names of isomorphisms defined in standard.iso or whatever isomorphism file has been included. There are, however, some other isomorphisms that are built into the implementation of Coccinelle and that can be disabled as well. Their names are given below. In each case, the text descibes the standard behavior. Using *disable-iso* with the given name disables this behavior.

- `optional_storage`: A SmPL function definition that does not specify any visibility (i.e., static or extern), or a SmPL variable declaration that does not specify any storage (i.e., auto, static, register, or extern), matches a function declaration or variable declaration with any visibility or storage, respectively.
- `optional_qualifier`: This is similar to `optional_storage`, except that here is it the qualifier (i.e., const or volatile) that does not have to be specified in the SmPL code, but may be present in the C code.
- `value_format`: Integers in various formats, e.g., 1 and 0x1, are considered to be equivalent in the matching process.
- `comm_assoc`: An expression of the form *exp bin_op* `...`, where *bin_op* is commutative and associative, is considered to match any top-level sequence of *bin_op* operators containing *exp* as the top-level argument.

The possible types of metavariable declarations are defined by the grammar rule below. Metavariables should occur at least once in the transformation immediately following their declaration. Fresh metavariables must only be used in + code. These properties are not expressed in the grammar, but are checked by a subsequent analysis. The metavariables are designated according to the kind of terms they can match, such as a statement, an identifier, or an expression. An expression metavariable can be further constrained by its type.

```
metadecl ::= fresh identifier ids ;
           | identifier COMMA_LIST(pmid_with_regexp) ;
           | identifier COMMA_LIST(pmid_with_not_eq) ;
           | parameter [list] ids ;
           | parameter list [ id ] ids ;
           | type ids ;
           | statement [list] ids ;
           | typedef ids ;
           | declarer name ids ;
           | declarer COMMA_LIST(pmid_with_regexp) ;
           | declarer COMMA_LIST(pmid_with_not_eq) ;
           | iterator name ids ;
           | iterator COMMA_LIST(pmid_with_regexp) ;
           | iterator COMMA_LIST(pmid_with_not_eq) ;
           | [local] idexpression [ctype] COMMA_LIST(pmid_with_not_eq) ;
           | [local] idexpression [{ctypes} *^*] COMMA_LIST(pmid_with_not_eq) ;
           | [local] idexpression *^+ COMMA_LIST(pmid_with_not_eq) ;
           | expression list ids ;
           | expression *^+ COMMA_LIST(pmid_with_not_eq) ;
           | expression COMMA_LIST(pmid_with_not_ceq) ;
           | expression list [ ident ] ids ;
           | ctype [ ] COMMA_LIST(pmid_with_not_eq) ;
           | ctype COMMA_LIST(pmid_with_not_ceq) ;
           | {ctypes} *^* COMMA_LIST(pmid_with_not_ceq) ;
           | {ctypes} *^* [ ] COMMA_LIST(pmid_with_not_eq) ;
           | constant [ctype] COMMA_LIST(pmid_with_not_eq) ;
           | constant [{ctypes} *^*] COMMA_LIST(pmid_with_not_eq) ;
           | position [any] COMMA_LIST(pmid_with_not_eq_mid) ;

ids                   ::= COMMA_LIST(pmid)
pmid                  ::= id
                        | mid
mid                   ::= rulename_id.id
pmid_with_regexp      ::= pmid ~= regexp
pmid_with_not_eq      ::= pmid [!= id]
                        | pmid [!= { COMMA_LIST(id) }]
pmid_with_not_ceq     ::= pmid [!= id_or_cst]
                        | pmid [!= { COMMA_LIST(id_or_cst) }]
id_or_cst             ::= id
                        | integer
pmid_with_not_eq_mid  ::= pmid [!= mid]
                        | pmid [!= { COMMA_LIST(mid) }]
```

Subsequently, we refer to arbitrary metavariables as metaid^{ty}, where *ty* indicates the *metakind* used in the declaration of the variable. For example, $\mathsf{metaid}^{\mathsf{Type}}$ refers to a metavariable that was declared using `type` and stands for any type.

The *ctype* and *ctypes* nonterminals are used by both the grammar of metavariable declarations and the grammar of transformations, and are defined on page 26.

9.3 Metavariables for scripts

Metavariables for scripts can only be inherited from transformation rules. In the spirit of scripting languages such as Python that use dynamic typing, metavariables for scripts do not include type declarations.

script_metavariables ::= `@ script:`*language* [`depends on` *dep*] `@` *script_metadecl** `@@`
| `@ initialize:`*language* `@`
| `@ finalize:`*language* `@`
language ::= `python`
script_metadecl ::= id `<<` rulename_id`.`id `;`

Currently, the only scripting language that is supported is Python. The set of available scripting languages may be extended at some point.

Script rules declared with `initialize` are run before the treatment of any file. Script rules declared with `finalize` are run when the treatment of all of the files has completed. There can be at most one of each per scripting language (thus currently at most one of each). Initialize and finalize script rules do not have access to SmPL metavariables. Nevertheless, a finalize script rule can access any variables initialized by the other script rules, allowing information to be transmitted from the matching process to the finalize rule.

9.4 Transformation

The transformation specification essentially has the form of C code, except that lines to remove are annotated with - in the first column, and lines to add are annotated with +. A transformation specification can also use *dots*, "`...`", describing an arbitrary sequence of function arguments or instructions within a control-flow path. Dots may be modified with a `when` clause, indicating a pattern that should not occur anywhere within the matched sequence. Finally, a transformation can specify a disjunction of patterns, of the form `(` pat_1 `|` ... `|` pat_n `)` where each `(`, `|` or `)` is in column 0 or preceded by `\`.

The grammar that we present for the transformation is not actually the grammar of the SmPL code that can be written by the programmer, but is instead the grammar of the slice of this consisting of the - annotated and the unannotated code (the context of the transformed lines), or the + annotated code and the unannotated code. For example, for parsing purposes, the following transformation is split into the two variants shown below and each is parsed separately.

```
 proc_info_func(...) {
   <...
-    hostno
+    hostptr->host_no
   ...>
 }
```

```
 proc_info_func(...) {
   <...
-    hostno
   ...>
 }
```

```
 proc_info_func(...) {
   <...
+    hostptr->host_no
   ...>
 }
```

Requiring that both slices parse correctly ensures that the rule matches syntactically valid C code and that it produces syntactically valid C code. The generated parse trees are then merged for use in the subsequent matching and transformation process.

The grammar for the minus or plus slice of a transformation is as follows:

```
transformation ::= include+
                 | OPTDOTSEQ(expr, when)
                 | OPTDOTSEQ(decl_stmt+, when)
                 | OPTDOTSEQ(fundecl, when)
include        ::= #include include_string
when           ::= when != when_code
                 | when = rule_elem_stmt
                 | when COMMA_LIST(any_strict)
                 | when true != expr
                 | when false != expr
when_code      ::= OPTDOTSEQ(decl_stmt+, when)
                 | OPTDOTSEQ(expr, when)
rule_elem_stmt ::= one_decl
                 | expr;
                 | return [expr];
                 | break;
                 | continue;
                 | \(rule_elem_stmt (\| rule_elem_stmt)+\)
any_strict     ::= any
                 | strict
                 | forall
                 | exists
```

```
OPTDOTSEQ(grammar_ds, when_ds) ::=
     [... [when_ds]] grammar_ds (... [when_ds] grammar_ds)* [... [when_ds]]
```

Lines may be annotated with an element of the set {−, +, ⋆} or the singleton ?, or one of each set. ? represents at most one match of the given pattern. ⋆ is used for semantic match, *i.e.*, a pattern that highlights the fragments annotated with ⋆, but does not perform any modification of the matched code. ⋆ cannot be mixed with − and +. There are some constraints on the use of these annotations:

- Dots, *i.e.* `...`, cannot occur on a line marked +.
- Nested dots, *i.e.*, dots enclosed in < and >, cannot occur on a line with any marking.

Each element of a disjunction must be a proper term like an expression, a statement, an identifier or a declaration. Thus, the rule on the left below is not a syntaxically correct SmPL rule. One may use the rule on the right instead.

```
@@
type T;
T b;
@@

(
 writeb(...,
|
 readb(
)
-(T)
 b)
```

```
@@
type T;
T b;
@@

(
read
|
write
)
  (...,
-  (T)
   b)
```

9.5 Types

```
ctypes                     ::= COMMA_LIST(ctype)
ctype                      ::= [const_vol] generic_ctype ⋆*
                            |  [const_vol] void ⋆+
                            |  (ctype (| ctype)*)
const_vol                  ::= const
                            |  volatile
generic_ctype              ::= ctype_qualif
                            |  [ctype_qualif] char
                            |  [ctype_qualif] short
                            |  [ctype_qualif] int
                            |  [ctype_qualif] long
                            |  [ctype_qualif] long long
                            |  double
                            |  float
                            |  [struct| union] id [{ struct_decl_list* }]
ctype_qualif               ::= unsigned
                            |  signed
struct_decl_list           ::= struct_decl_list_start
struct_decl_list_start     ::= struct_decl
                            |  struct_decl struct_decl_list_start
                            |  ...  [when != struct_decl]† [continue_struct_decl_list]
continue_struct_decl_list  ::= struct_decl struct_decl_list_start
                            |  struct_decl
struct_decl                ::= ctype d_ident;
                            |  fn_ctype (⋆ d_ident) (PARAMSEQ(name_opt_decl, ε));)
                            |  [const_vol] id d_ident;
d_ident                    ::= id [[expr]]*
fn_ctype                   ::= generic_ctype ⋆*
                            |  void ⋆*
name_opt_decl              ::= decl
                            |  ctype
                            |  fn_ctype
```

† The optional `when` construct ends at the end of the line.

9.6 Function declarations

```
fundecl   ::= [fn_ctype] funinfo* funid ([PARAMSEQ(param, ε)]) { [stmt_seq] }
funproto  ::= [fn_ctype] funinfo* funid ([PARAMSEQ(param, ε)]);
funinfo   ::= inline
            | storage
storage   ::= static
            | auto
            | register
            | extern
funid     ::= id
            | metaid^Id
param     ::= type id
            | metaid^Param
            | metaid^ParamList
decl      ::= ctype id
            | fn_ctype (* id) (PARAMSEQ(name_opt_decl, ε))
            | void
            | metaid^Param

PARAMSEQ(gram_p, when_p) ::= COMMA_LIST(gram_p | ...[when_p])
```

9.7 Declarations

```
decl_var     ::= common_decl
               | [storage] ctype COMMA_LIST(d_ident) ;
               | [storage] [const_vol] id COMMA_LIST(d_ident) ;
               | [storage] fn_ctype ( * d_ident ) ( PARAMSEQ(name_opt_decl, ε) ) = initialize ;
               | typedef ctype typedef_ident ;
one_decl     ::= common_decl
               | [storage] ctype id;
               | [storage] [const_vol] id d_ident ;
common_decl  ::= ctype;
               | funproto
               | [storage] ctype d_ident = initialize ;
               | [storage] [const_vol] id d_ident = initialize ;
               | [storage] fn_ctype ( * d_ident ) ( PARAMSEQ(name_opt_decl, ε) ) ;
               | decl_ident ( [COMMA_LIST(expr)] ) ;
initialize   ::= dot_expr
               | { [COMMA_LIST(dot_expr)] }
decl_ident   ::= DeclarerId
               | metaid^Declarer
```

9.8 Statements

The first rule *statement* describes the various forms of a statement. The remaining rules implement the constraints that are sensitive to the context in which the statement occurs: *single_statement* for a context in which only one statement is allowed, and *decl_statement* for a context in which a declaration, statement, or sequence thereof is allowed.

```
stmt         ::= include
               | metaid^Stmt
               | expr;
               | if (dot_expr) single_stmt [else single_stmt]
               | for ([dot_expr]; [dot_expr]; [dot_expr]) single_stmt
               | while (dot_expr) single_stmt
               | do single_stmt while (dot_expr);
               | iter_ident (dot_expr*) single_stmt
               | switch ([dot_expr]) {case_line* }
               | return [dot_expr];
               | { [stmt_seq] }
               | NEST(decl_stmt+, when)
               | NEST(expr, when)
               | break;
               | continue;
               | id:
               | goto id;
               | {stmt_seq }
single_stmt  ::= stmt
               | OR(stmt)
decl_stmt    ::= metaid^StmtList
               | decl_var
               | stmt
               | OR(stmt_seq)
stmt_seq     ::= decl_stmt* [DOTSEQ(decl_stmt+, when) decl_stmt*]
               | decl_stmt* [DOTSEQ(expr, when) decl_stmt*]
case_line    ::= default : stmt_seq
               | case dot_expr : stmt_seq
iter_ident   ::= IteratorId
               | metaid^Iterator

OR(gram_o)                 ::= ( gram_o (|gram_o)* )
DOTSEQ(gram_d, when_d)     ::= ...[when_d] (gram_d ...[when_d])*
NEST(gram_n, when_n)       ::= <...[when_n] gram_n (...[when_n] gram_n)* ...>
                             | <+...[when_n] gram_n (...[when_n] gram_n)* ...+>
```

OR is a macro that generates a disjunction of patterns. The three tokens (, |, and) must appear in the leftmost column, to differentiate them from the parentheses and bit-or tokens that can appear within expressions (and cannot appear in the leftmost column). These token may also be preceded by \ when they are used in an other column. These tokens are furthermore different from (, |, and), which are part of the grammar metalanguage.

9.9 Expressions

A nest or a single ellipsis is allowed in some expression contexts, and causes ambiguity in others. For example, in a sequence `...`*expr* `...`, the nonterminal *expr* must be instantiated as an explicit C-language expression, while in an array reference, $expr_1$ `[` $expr_2$ `]`, the nonterminal $expr_2$, because it is delimited by brackets, can be also instantiated as `...`, representing an arbitrary expression. To distinguish between the various possibilities, we define three nonterminals for expressions: *expr* does not allow either top-level nests or ellipses, *nest_expr* allows a nest but not an ellipsis, and *dot_expr* allows both. The EXPR macro is used to express these variants in a concise way.

```
expr          ::= EXPR(expr)
nest_expr     ::= EXPR(nest_expr)
                |  NEST(nest_expr, exp_whencode)
dot_expr      ::= EXPR(dot_expr)
                |  NEST(dot_expr, exp_whencode)
                |  ... [exp_whencode]
EXPR(exp)     ::= exp assign_op exp
                |  exp++
                |  exp−
                |  unary_op exp
                |  exp bin_op exp
                |  exp ? dot_expr : exp
                |  (type) exp
                |  exp [dot_expr]
                |  exp . id
                |  exp -> id
                |  exp([PARAMSEQ(arg, exp_whencode)])
                |  id
                |  metaid^Exp
                |  metaid^Const
                |  const
                |  (dot_expr)
                |  OR(exp)
arg           ::= nest_expr
                |  metaid^ExpList
exp_whencode  ::= when != expr
assign_op     ::= = | -= | += | *= | /= | %=
                |  &= | |= | ^= | <<= | >>=
bin_op        ::= * | / | % | + | -
                |  <<| >>| ^ | & | |
                |  < | > | <= | >= | == | != | && | ||
unary_op      ::= ++ | - | & | * | + | - | !
```

9.10 Constant, Identifiers and Types for Transformations

```
const         ::= string
                |  [0-9]+
                |  ...
string        ::= "[^"]*"
id            ::= id | metaid^Id
typedef_ident ::= id | metaid^Type
type          ::= ctype | metaid^Type
pathToIsoFile ::= <.*>
regexp        ::= "[^"]*"
```

Chapter 10

spatch command line options

10.1 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.

10.2 Selecting and parsing the semantic patch

10.2.1 Standalone options

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

10.2.2 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; the last two are deprecated.

10.2.3 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.

-track_iso Gather information about isomorphism usage.

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

10.2.4 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.

10.3 Selecting and parsing the C files

10.3.1 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`.

◇ **-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.

10.3.2 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.

◇ **-dir** Specify a directory containing C files to process. 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

10.3.3 Parsing C files

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

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

◇ **-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.

-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.

◆ **-macro_file_builtins ⟨file⟩** Builtin macro definitions to be taken into account when parsing the C files.

◇ **-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

◆ **-all_includes, -local_includes, -no_includes** These options control which include files mentioned in a C file are taken into account. **-all_includes** indicates that all included files 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 the directory in which to find non-local include files. This option should be used only once, as each use will overwrite the preceding one.

◇ **-relax_include_path** This option causes the search for local include files to consider the directory specified using **-I** if the included file is not found in the current directory.

10.4 Application of the semantic patch to the C code

10.4.1 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.

10.4.2 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**.

10.4.3 Actions during the application of the semantic patch

◇ **-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.

-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**.

10.5 Generation of the result

Normally, the only output is a diff printed to standard output.

⬙ **-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⟩** The output file.

◇ **-inplace** Modify the input file.

◇ **-outplace** Store modifications in a .cocci_res file.

◇ **-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.

◇ **-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.

-patch Deprecated option.

10.6 Other options

10.6.1 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.

10.6.2 Help

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

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

10.6.3 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.

10.6.4 Miscellaneous

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

-pad, -hrule ⟨dir⟩, -xxx, -l1