Coccinelle

User's manual
release 1.0.6

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

July 13, 2018

Contents

Foreword

This manual documents the release 1.0.6 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 and a wiki website.

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 a specific language called SmPL, for Semantic Patch Language, which as its name suggests is very close to the syntax of a patch, but does not work at a line level, as traditional patches do, but rather at higher, semantic level.

Here is an example of a simple program transformation. To replace every call to `foo` of any expression x by a call to `bar`, create a semantic patch file `ex1.cocci` (semantic patches usually end 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
```

Coccinelle primarily targets ANSI C, and supports some GCC extensions. It has only partial support for K&R C. K&R function declarations are only recognized if the parameter declarations are indented. Furthermore, the parameter names are subsequently considered to be type names, due to confusion with function prototypes, in which a name by itself is indeed the name of a type.

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

```
#DEFINE TEST "foo"

printf("foo");

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

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

  foo();
}
```

Semantic patch

```
@@

@@


- foo()
+ bar()
```

After

```
#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 an 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.

One 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 effect, 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 to 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 previous 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 several positions are matched by a single position variable and 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 expressions

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

This section presents some tips and tricks for using Coccinelle.

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 square bracket that is surrounded by spaces in the description of a term should appear explicitly in the term, as in an array reference. On the other hand, square brackets that surround some other term indicate that the presence of that term is optional.

An 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	::=	`include` *string*
	|	`using` *string*
	|	`using` *pathToIsoFile*
	|	`virtual` id (`,` id)*
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 comment.

`virtual` keyword is used to declare virtual rules. Virtual rules may be subsequently used as a dependency for the rules in the SmPL file. Whether a virtual rule is defined or not is controlled by the `-D` option on the command line.

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`]
dep	::=	id
	|	`!`id
	|	`!(`*dep*`)`
	|	`ever` id
	|	`never` id
	|	*dep* `&&` *dep*
	|	*dep* `||` *dep*
	|	`file in` *string*
	|	`(`*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 describes 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 it is 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.

- `optional_attributes`: This is similar to `optional_attributes`, except that here is it an attribute (e.g., __init) that does not have to be specified in the SmPL code, but may be present in the C code. **Note that this isomorphism is currently useless, because matching of attributes is not supported, due to the difficulty of parsing attributes in C code.**

- `value_format`: Integers in various formats, e.g., 1 and 0x1, are considered to be equivalent in the matching process.

- `optional_declarer_semicolon`: Some declarers (top-level terms that look like function calls but serve to declare some variable) don't require a semicolon. This isomorphism allows a SmPL declarer with a semicolon to match such a C declarer, if no transformation is specified on the SmPL semicolon.

- `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.

- `prototypes`: A rule for transforming a function prototype is generated when a function header changes.

The `depends on` clause indicates conditions under which a semantic patch rule should be applied. Most of these conditions relate to the success or failure of other rules, which may be virtual rules. Giving the name of a rule implies that the current rule is applied if the named rule has succeeded in matching in the current environment. Giving `ever` followed by a rule name implies that the current rule is applied if the named rule has succeeded in matching in any environment. Analogously, `never` means that the named rule should have succeeded in matching in no environment. The boolean and, or and negation operators combine these declarations in the usual way. The declaration `file in` checks that the code being processed comes from the mentioned file, or from a subdirectory. The declaration `file` `in` is only allowed on SmPL code-matching rules. Script rules are not applied to any code in particular, and thus it doesn't make sense to check on the file being considered.

The possible types of metavariable declarations are defined by the grammar rule below. Metavariables should occur at least once in the transformation code immediately following their declaration. Fresh identifier 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. A declaration metavariable matches the declaration of one or more variables, all sharing the same type specification (*e.g.*, `int a,b,c=3;`). A field metavariable does the same, but for structure fields. In the minus code, a statement list metavariable can only appear as a complete function body or as the complete body of a sequence statement. In the plus code, a statement list metavariable can occur anywhere a statement list is allowed, i.e., including as an element of another statement list.

```
metadecl            ::= metavariable ids ;
                      | fresh identifier ids ;
                      | identifier COMMA_LIST(pmid_with_regexp) ;
                      | identifier COMMA_LIST(pmid_with_virt_or_not_eq) ;
                      | parameter [list] ids ;
                      | parameter list [ id ] ids ;
                      | parameter list [ const ] ids ;
                      | identifier [list] ids ;
                      | identifier list [ id ] ids ;
                      | identifier list [ const ] ids ;
                      | type ids ;
                      | statement [list] ids ;
                      | declaration ids ;
                      | field [list] ids ;
                      | typedef ids ;
                      | attribute 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 | global] idexpression [ctype] COMMA_LIST(pmid_with_not_eq) ;
                      | [local | global] idexpression [{ctypes} ⋆*] COMMA_LIST(pmid_with_not_eq) ;
                      | [local | global] idexpression ⋆+ COMMA_LIST(pmid_with_not_eq) ;
                      | expression list ids ;
                      | expression ⋆+ COMMA_LIST(pmid_with_not_eq) ;
                      | expression enum ⋆* COMMA_LIST(pmid_with_not_eq) ;
                      | expression struct ⋆* COMMA_LIST(pmid_with_not_eq) ;
                      | expression union ⋆* COMMA_LIST(pmid_with_not_eq) ;
                      | expression COMMA_LIST(pmid_with_not_ceq) ;
                      | expression list [ id ] ids ;
                      | expression list [ const ] 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) ;
                      | symbol ids;
                      | format ids;
                      | format list [ id ] ids ;
                      | format list [ const ] ids ;
                      | assignment operator COMMA_LIST(assignopdecl) ;
                      | binary operator COMMA_LIST(binopdecl) ;
assignopdecl        ::= id [ = assignop_contraint]
assignop_contraint  ::= {COMMA_LIST(assign_op)}
                      | assign_op
binopdecl           ::= id [ = binop_contraint]
binop_contraint     ::= {COMMA_LIST(bin_op)}
                      | bin_op
```

A metavariable declaration local idexpression v means that v is restricted to be a local variable. If it should just be a variable, but not necessarily a local one, then drop local. A more complex description of a location, such as a->b is considered to be an expression, not an idexpression.

Constant is for constants, such as 27. But it also considers an identifier that is all capital letters (possibly containing numbers) as a constant as well, because the names given to macros in Linux usually have this form.

An identifier is the name of a structure field, a macro, a function, or a variable. It is the name of something rather than an expression that has a value. But an identifier can be used in the position of an expression as well, where it represents a variable.

It is possible to specify that an expression list or a parameter list metavariable should match a specific number of expressions or parameters.

An identifier list is only used for the parameter list of a macro. It is possible to specify its length.

It is possible to specify some information about the definition of a fresh identifier. See the wiki.

A symbol declaration specifies that the provided identifiers should be considered C identifiers when encountered in the body of the rule. Identifiers in the body of the rule that are not declared explicitly are by default considered symbols, thus symbol declarations are optional. It is not required, but it will not cause a parse error, to redeclare a name as a symbol. A name declared as a symbol can, however, be redeclared as another metavariable. It will be considered to be a metavariable in such rules, and will revert to being a symbol in subsequent rules. These conditions also apply to iterator names and declarer names.

An attribute declaration indicates a name that should be considered to be an attribute. It is not possible to match or remove an attribute, only to add one.

A position metavariable is used by attaching it using `@` to any token, including another metavariable. Its value is the position (file, line number, etc.) of the code matched by the token. It is also possible to attach expression, declaration, type, initialiser, and statement metavariables in this manner. In that case, the metavariable is bound to the closest enclosing expression, declaration, etc. If such a metavariable is itself followed by a position metavariable, the position metavariable applies to the metavariable that it follows, and not to the attached token. This makes it possible to get eg the starting and ending position of `f(...)`, by writing `f(...)@E@p`, for expression metavariable `E` and position metavariable `p`. This attachment notation for metavariables of type other than position can also be expressed with a conjunction, but the @ notation may be more concise.

When used, a format or format list metavariable must be enclosed by a pair of `@`s. A format metavariable matches the format descriptor part, i.e., `2x` in `%2x`. A format list metavariable matches a sequence of format descriptors as well as the text between them. Any text around them is matched as well, if it is not matched by the surrounding text in the semantic patch. Such text is not partially matched. If the length of the format list is specified, that indicates the number of matched format descriptors. It is also possible to use `...` in a format string, to match a sequence of text fragments and format descriptors. This only takes effect if the format string contains format descriptors. Note that this makes it impossible to require `...` to match exactly in a string, if the semantic patch string contains format descriptors. If that is needed, some processing with a scripting language would be required. And example for the use of string format metavariables is found in demos/format.cocci.

Assignment (resp. binary) operator metavariables match any assignment (resp. binary) operator. The list of operators that can be matched can be restricted by adding an operator constraint, i.e. a list of accepted operators.

Other kinds of metavariables can also be attached using `@` to any token. In this case, the metavariable floats up to the enclosing appropriate expression. For example, `3 +@E 4`, where `E` is an expression metavariable binds `E` to `3 + 4`. A particular case is `Ps@Es`, where `Ps` is a parameter list and `Es` is an expression list. This pattern matches a parameter list, and then matches `Es` to the list of expressions, ie a possible argument list, represented by the names of the parameters. Another particular case is `E@S`, where `E` is any expression and `S` is a statement metavariable. `S` matches the closest enclosing statement, which may be more than what is matches by the semantic match pattern itself.

Matching of various kinds of format strings within strings is supported. With the `-ibm` option, matching of decimal format declarations is supported, but the length and precision arguments are not interpreted. Thus it is not possible to match metavariables in these fields. Instead, the entire format is matched as a single string.

ids	::=	*COMMA_LIST*(*pmid*)
pmid	::=	id
	|	*mid*
mid	::=	rulename_id . id
pmid_with_regexp	::=	*pmid* `=~` *regexp*
	|	*pmid* `!~` *regexp*
pmid_with_not_eq	::=	*pmid* [`!=` *id_or_meta*]
	|	*pmid* [`!=` `{` *COMMA_LIST*(*id_or_meta*) `}`]
pmid_with_virt_or_not_eq	::=	`virtual`.id
	|	*pmid_with_not_eq*
pmid_with_not_ceq	::=	*pmid* [`!=` *id_or_cst*]
	|	*pmid* [`!=` `{` *COMMA_LIST*(*id_or_cst*) `}`]
id_or_cst	::=	id
	|	integer
id_or_meta	::=	id
	|	rulename_id . id
pmid_with_not_eq_mid	::=	*pmid* [*ANDAND_LIST*(*pos_constraint*)]
pos_constraint	::=	`!=` *mid*
	|	`!=` `{` *COMMA_LIST*(*mid*) `}`
	|	`:` `script:ocaml` `(`*COMMA_LIST*(*mid*)`)` `{`*expr* `}`
ANDAND_LIST(*X*)	::=	*X*
	|	*X* `&&` *ANDAND_LIST*(*X*)

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

`metavariable` declares a metavariable for which the parser tried to figure out the metavariable type based on the usage context. Such a metavariable must be used consistently. These metavariables cannot be used in all contexts; specifically, they cannot be used in context that would make the parsing ambiguous. Some examples are the leftmost term of an expression, such as the left-hand side of an assignment, or the type in a variable declaration. These restrictions may seems somewhat arbitrary from the user's point of view. Thus, it is better to use metavariables with metavariable types. If Coccinelle is given the argument `-parse_cocci`, it will print information about the type that is inferred for each metavariable.

The *ctype* and *ctypes* nonterminals are used by both the grammar of metavariable declarations and the grammar of transformations, and are defined on page 35.

An identifier metavariable with `virtual` as its "rule name" is given a value on the command line. For example, if a semantic patch contains a rule that declares an identifier metavariable with the name `virtual.alloc`, then the command line could contain `-D alloc=kmalloc`. There should not be space around the `=`. An example is in `demos/vm.cocci` and `demos/vm.c`.

It is possible to give an identifier metavariable a list of constraints that it should or should not be equal to. If the constraint is a list of (unquoted) strings, then the value of the metavariable should be the same as one of the strings, in the case of an equality constraint, or different from all of the strings, in the case of an inequality constraint. It is also possible to include inherited identifier metavariables among the constraints. In the case of a positive constraint, things work in the same way, but not with respect to the inherited value of the metavariable. On the other hand, an inequality constraint does not work so well, because the only value available is the one available in the current environment. If the proposed value is different from the one in the current environment, but perhaps the same as the one in some other environment, the match will still succeed.

Position metavariables can be associated with constraints implemented as OCaml script code. The code must have the form of a single C expression, typically a function call with a tuple of arguments. This expression must have type bool. The script code can be parameterized by any inherited metavariables. It is implicitly parameterized by the metavariable being declared. In the script, the inherited variable parameters are referred to by their variable names, without the associated rule name. The variable being declared is also referenced by its name. All parameters, except position variables, have their string representation. An example is in demos/poscon.cocci.

A declaration of a name as a typedef extends through the rest of the semantic patch. It is not required, but it will not cause a parse error, to redeclare a name as a typedef. A name declared as a typedef can, however, be redeclared as another metavariable. It will be considered to be a metavariable in such rules, and will revert to being a typedef in subsequent rules.

Warning: Each metavariable declaration causes the declared metavariables to be immediately usable, without any inheritance indication. Thus the following are correct:

```
@@
type r.T;
T x;
@@

[...] // some semantic patch code
```

```
@@
r.T x;
type r.T;
@@

[...] // some semantic patch code
```

But the following is not correct:

```
@@
type r.T;
r.T x;
@@

[...] // some semantic patch code
```

This applies to position variables, type metavariables, identifier metavariables that may be used in specifying a structure type, and metavariables used in the initialization of a fresh identifier. In the case of a structure type, any identifier metavariable indeed has to be declared as an identifier metavariable in advance. The syntax does not permit `r.n` as the name of a structure or union type in such a declaration.

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 [rulename] [depends on dep] @ script_metadecl* @@
                       | @ initialize:language [depends on dep] @ script_virt_metadecl* @@
                       | @ finalize:language [depends on dep] @ script_virt_metadecl* @@
language             ::= python
                       | ocaml
script_metadecl      ::= id << rulename_id.id ;
                       | id << rulename_id.id = "..." ;
                       | id << rulename_id.id = [] ;
                       | id ;
script_virt_metadecl ::= id << virtual.id ;
```

Currently, the only scripting languages that are supported are Python and OCaml, indicated using `python` and `ocaml`, respectively. 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.

Initialize and finalize rules do have access to virtual metavariables, using the usual syntax. As for other scripting language rules, the rule is not run (and essentially does not exist) if some of the required virtual metavariables are not bound. In ocaml, a warning is printed in this case. An example is found in demos/initvirt.cocci.

A script metavariable that does not specify an origin, using «, is newly declared by the script. This metavariable should be assigned to a string and can be inherited by subsequent rules as an identifier. In Python, the assignment of such a metavariable x should refer to the metavariable as `coccinelle.`x. Examples are in the files `demos/pythontococci.cocci` and `demos/camltococci.cocci`.

In an OCaml script, the following extended form of *script_metadecl* may be used:

```
script_metadecl' ::= (id,id) << rulename_id.id ;
                   | id << rulename_id.id ;
                   | id ;
```

In a declaration of the form `(id,id) << rulename_id.id ;`, the left component of `(id,id)` receives a string representation of the value of the inherited metavariable while the right component receives its abstract syntax tree. The file `parsing_c/ast_c.ml` in the Coccinelle implementation gives some information about the structure of the abstract syntax tree. Either the left or right component may be replaced by _, indicating that the string representation or abstract syntax trees representation is not wanted, respectively.

The abstract syntax tree of a metavariable declared using `metavariable` is not available.

Script metavariables can have default values. This is only allowed if the abstract syntax tree of the metavariable is not requested. The default value of a position metavariable is written as `[]`. The default value of any other kind of metavariable is a string. There is no control that the string actually represents the kind of term represented by the metavariable. Normally, a script rule is only applied if all of the metavariables have values. If default values are provided, then the script rule is only applied if all of the metavariables for which there are no default values have values. See demos/defaultscript.cocci for examples of the use of this feature.

9.4 Control Flow

Rules describe a property that Coccinelle must match, and when the property described is matched the rule is considered successful. One aspect that is taken into account in determining a match is the program control flow. A control flow describes a possible run time path taken by a program.

9.4.1 Basic dots

When using Coccinelle, it is possible to express matches of certain code within certain types of control flows. Ellipses ("...") can be used to indicate to Coccinelle that anything can be present between consecutive statements. For instance the following SmPL patch tells Coccinelle that rule r0 wishes to remove all calls to function c().

```
@r0@
@@

-c();
```

The context of the rule provides no other guidelines to Coccinelle about any possible control flow other than this is a statement, and that c() must be called. We can modify the required control flow required for this rule by providing additional requirements and using ellipses in between. For instance, if we only wanted to remove calls to c() that also had a prior call to foo() we'd use the following SmPL patch:

```
@r1@
@@

foo()
...
-c();
```

9.4.2 Dot variants

There are two possible modifiers to the control flow for ellipses, one (<... ...>) indicates that matching the pattern in between the ellipses is optional, and another (<+... ...+>) indicates that the pattern in between the ellipses must be matched at least once, on some control-flow path. In the latter, the + is intended to be reminiscent of the + used in regular expressions. For instance, the following SmPL patch tells Coccinelle to remove all calls to c(). if foo() is present at least once since the beginning of the function.

```
@r2@
@@

<+...
foo()
...+>
-c();
```

Alternatively, the following indicates that foo() is allowed but optional. This case is typically most useful when all occurrences, if any, of foo() prior to c() should be transformed.

```
@r3@
@@

<...
foo()
...>
-c();
```

9.4.3 An example

Let's consider some sample code to review: flow1.c.

```

int main(void)
{
	int ret, a = 2;

	a = foo(a);
	ret = bar(a);
	c();

	return ret;
}
```

Applying the SmPL rule r0 to flow1.c would remove the c() line as the control flow provides no specific context requirements. Applying rule r1 would also succeed as the call to foo() is present. Likewise rules r2 and r3 would also succeed. If the foo() call is removed from flow1.c only rules r0 and r3 would succeed, as foo() would not be present and only rules r0 and r3 allow for foo() to not be present.

One way to describe code control flow is in terms of McCabe cyclomatic complexity. The program flow1.1 has a linear control flow, it has no branches. The main routine has a McCabe cyclomatic complexity of 1. The McCabe cyclomatic complexity can be computed using `pmccabe` (https://www.gnu.org/software/complexity/manual/-html_node/pmccabe-parsing.html).

```
pmccabe /flow1.c
1          1          5          1          10         flow1.c(1): main
```

Since programs can use branches, often times you may also wish to annotate requirements for control flows in consideration for branches, for when the McCabe cyclomatic complexity is > 1. The following program, flow2.c, enables the control flow to diverge on line 7 due to the branch, if (a) – one control flow possible is if (a) is true, another when if (a) is false.

```
1  int main(void)
2  {
3          int ret, a = 2;
4
5          a = foo(a);
6          ret = bar(a);
7          if (a)
8                  c();
9
10         return ret;
11 }
```

This program has a McCabe cyclomatic complexity of 2.

```
pmccabe flow2.c
2          2          6          1          11         flow2.c(1): main
```

Using the McCabe cyclomatic complexity is one way to get an idea of the complexity of the control graph for a function, another way is to visualize all possible paths. Coccinelle provides a way to visualize control flows of programs, this however requires `dot` (http://www.graphviz.org/) and `gv` to be installed (typically provided by a package called graphviz). To visualize control flow or a program using Coccinelle you use:

spatch –control-flow-to-file flow1.c spatch –control-flow-to-file flow2.c

Behind the scenes this generates a dot file and uses gv to generate a PDF file for viewing. To generate and inspect these manually you can use the following:

spatch –control-flow-to-file flow2.c dot -Tpdf flow1:main.dot > flow1.pdf

By default properties described in a rule must match all control flows possible within a code section being inspected by Coccinelle. So for instance, in the following SmPL patch rule r1 would match all the control flow possible on flow1.c as its linear, however it would not match the control possible on flow2.c. The rule r1 would not be successful in flow2.c

```
@r1@
@@

foo()
...
-c();
```

The default control flow can be modified by using the keyword "exists" following the rule name. In the following SmPL patch the rule r2 would be successful on both flow1.c and flow2.c

```
@r2 exists@
@@

foo()
...
-c();
```

If the rule name is followed by the "forall" keyword, then all control flow paths must match in order for the rule to succeed. By default when a semantic patch has "-" and "+", or when it has no annotations at all and only script code, ellipses ("...") use the forall semantics. And when the semantic patch uses the context annotation ("*"), the ellipses ("...") uses the exists semantics. Using the keyword "forall" or "exists" in the rule header affects all ellipses ("...") uses in the rule. You can also annotate each ellipses ("...") with "when exists" or "when forall" individually.

Rules can also be not be successful if requirements do not match when a rule name is followed by "depends on XXX". When "depends on" is used it means the rule should only apply if rule XXX matched with the current metavariable environment. Alternatively, "depends on ever XXX" can be used as well, this means this rule should apply if rule XXX was ever matched at all. A counter to this use is "depends on never XXX", which means that this rule should apply if rule XXX was never matched at all.

9.5 Transformation

Coccinelle semantic patches are able to transform C code.

9.5.1 Basic transformations

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. Implicitly, "`...`" matches the shortest path between something that matches the pattern before the dots (or the beginning of the function, if there is nothing before the dots) and something that matches the pattern after the dots (or the end of the function, if there is nothing after the dots). Dots may be modified with a `when` clause, indicating a pattern that should not occur anywhere within the matched sequence. `when any` removes the aforementioned constraint that "`...`" matches the shortest path. 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 `\`. Similarly, a transformation can specify a conjunction of patterns, of the form `(` pat_1 `&` `...` `&` pat_n `)` where each `(`, `&` or `)` is in column 0 or preceded by `\`. All of the patterns must be matched at the same place in the control-flow graph.

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*(*top*`,` *when*)
include	::=	`#include` include_string
top	::=	*expr*
	|	*decl_stmt*$^+$
	|	*fundecl*
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 $\{-, +, \star\}$ or the singleton `?`, or one of each set. `?` represents at most one match of the given pattern, ie a match of the pattern is optional. $\star$ is used for semantic match, *i.e.*, a pattern that highlights the fragments annotated with $\star$, but does not perform any modification of the matched code. $\star$ 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 marked +.

An #include may be followed by "...", <...> or simply .... With either quotes or angle brackets, it is possible to put a partial path, ending with ..., such as <include/...>, or to put a complete path. A #include with ... matches any include, with either quotes or angle brackets. Partial paths or complete are not allowed in the latter case. Something that is added before an include will be put before the last matching include that is not under an ifdef in the file. Likewise, something that is added after an include will be put after the last matching include that is not under an ifdef in the file.

Each element of a disjunction must be a proper term like an expression, a statement, an identifier or a declaration. The constraint on a conjunction is similar. Thus, the rule on the left below is not a syntactically 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)
```

Some kinds of terms can only appear in + code. These include comments, ifdefs, and attributes (__attribute__((...))).

9.5.2 Advanced transformations

You may run into the situation where a semantic patch needs to add several disjoint terms at the same place in the code. Coccinelle does not know in which order these terms should appear, and thus gives an "already tagged token" error in this situation. If you are sure that order does not matter you can use the optional double addition token ++ to indicate to Coccinelle that it may add things in any order. This may be for instance safe in situations such as extending a data structure with more members, based on existing members of the data structure. The following rule helps to extend a data structure with a respective float for a present int. If there is only one int field in the data structure, this semantic patch works well with the simple +.

```
@simpleplus@
identifier x,v;
fresh identifier xx = v ## "_float";
@@

struct x {
+        float xx;
         ...
         int v;
         ...
}
```

This semantic patch works fine, for example, on the following code (plusplus1.c):

```
struct x {
        int z;
```

```
        char b;
};
```

If however there are multiple int fields tokens that Coccinelle can transform, order cannot be guaranteed for how Coccinelle makes additions. If you are sure order does not matter for the transformation you may use ++ instead, as follows:

```
@plusplus@
identifier x,v;
fresh identifier xx = v ## "_float";
@@

struct x {
++      float xx;
        ...
        int v;
        ...
}
```

This rule would work against a file plusplus2.c that has three int fields:

```
struct x {
        int z;
        int a;
        char b;
        int c;
        int *d;
};
```

A possible result is as shown below. The precise order of the float fields is however not guaranteed with respect to each other:

```
struct x {
        float a_float;
        float c_float;
        float z_float;
        int z;
        int a;
        char b;
        int c;
        int *d;
};
```

If you used simpleplus rule on plusplus2.c you would end up with an “already tagged token” error due to the ordering considerations explained in this section.

9.6 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] short int
                            |  [ctype_qualif] int
                            |  [ctype_qualif] long
                            |  [ctype_qualif] long int
                            |  [ctype_qualif] long long
                            |  [ctype_qualif] long long int
                            |  double
                            |  long double
                            |  float
                            |  size_t
                            |  ssize_t
                            |  ptrdiff_t
                            |  enum id { PARAMSEQ(dot_expr, exp_whencode) [,] }
                            |  [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.

9.7 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
	|	$\text{metaid}^{\text{Id}}$
	|	*OR*(*stmt*)
param	::=	*type* id
	|	$\text{metaid}^{\text{Param}}$
	|	$\text{metaid}^{\text{ParamList}}$
	|	`......`
decl	::=	*ctype* *id*
	|	*fn_ctype* `(*` *id*`)` `(`*PARAMSEQ*(*name_opt_decl*`,` ε)`)`
	|	`void`
	|	$\text{metaid}^{\text{Param}}$

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

To match a function it is not necessary to provide all of the annotations that appear before the function name. For example, the following semantic patch:

```
@@
@@

foo() { ... }
```

matches a function declared as follows:

```
static int foo() { return 12; }
```

This behavior can be turned off by disabling the `optional_storage` isomorphism. If one adds code before a function declaration, then the effect depends on the kind of code that is added. If the added code is a function definition or CPP code, then the new code is placed before all information associated with the function definition, including any comments preceding the function definition. On the other hand, if the new code is associated with the function, such as the addition of the keyword `static`, the new code is placed exactly where it appears with respect to the rest of the function definition in the semantic patch. For example,

```
@@
@@

+ static
foo() { ... }
```

causes static to be placed just before the function name. The following causes it to be placed just before the type

```
@@
type T;
@@

+ static
T foo() { ... }
```

It may be necessary to consider several cases to ensure that the added code is placed in the right position. For example, one may need one pattern that considers that the function is declared `inline` and another that considers that it is not.

Varargs are written in C using `...`. Unfortunately, this notation is already used in the semantic patch language. A pattern for a varargs parameter is written as a sequence of 6 dots.

The C parser allows functions that have no return type, and assumes that the return type is `int`. The support for parsing such functions is limited. In particular, the parameter list must contain a type for each parameter, and may not contain varargs.

For a function prototype, unlike a function definition, a specification of the return type is obligatory.

9.8 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*
	|	metaid$^{\text{Initialiser}}$
	|	`{` [*COMMA_LIST*(*init_list_elem*)] `}`
init_list_elem	::=	*dot_expr*
	|	*designator* `=` *initialize*
	|	metaid$^{\text{Initialiser}}$
	|	metaid$^{\text{InitialiserList}}$
	|	*id* `:` *dot_expr*
designator	::=	`.` *id*
	|	`[` *dot_expr* `]`
	|	`[` *dot_expr* `...` *dot_expr* `]`
decl_ident	::=	DeclarerId
	|	metaid$^{\text{Declarer}}$

An initializer for a structure can be ordered or unordered. It is considered to be unordered if there is at least one key-value pair initializer, e.g., `.x = e`.

A declaration can have *e.g.* the form `register x;`. In this case, the variable implicitly has type int, and SmPL code that declares an int variable will match such a declaration. On the other hand, the implicit int type has no position. If the SmPL code tries to record the position of the type, the match will fail.

9.9 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         ::= directive
               | 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 }
directive    ::= include
               | #define id [top]
               | #define id (PARAMSEQ(id, ε)) [top]
               | #undef id
               | #pragma id id+
               | #pragma id (PARAMSEQ(expr, ε))
               | #pragma id ...
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.

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

9.11 Constants, Identifiers and Types for Transformations

const	::=	*string*
	$\mid$	`[0-9]+`
	$\mid$	`...`
string	::=	`"[^"]*"`
id	::=	id $\mid$ metaid$^{\mathsf{Id}}$ $\mid$ *OR*(*stmt*)
typedef_ident	::=	id $\mid$ metaid$^{\mathsf{Type}}$
type	::=	*ctype* $\mid$ metaid$^{\mathsf{Type}}$
pathToIsoFile	::=	`<.*>`
regexp	::=	`"[^"]*"`

9.12 Comments and preprocessor directives

A `//` or `/* */` comment that is annotated with + in the leftmost column is considered to be added code. A `//` or `/* */` comment without such an annotation is considered to be a comment about the SmPL code, and thus is not matched in the C code.

The following preprocessor directives can likewise be added. They cannot be matched against. The entire line is added, but it is not parsed.

- `if`
- `ifdef`
- `ifndef`
- `else`
- `elif`
- `endif`
- `error`
- `line`

9.13 Command-line semantic match

It is possible to specify a semantic match on the spatch command line, using the argument `-sp`. In such a semantic match, any token beginning with a capital letter is assumed to be a metavariable of type `metavariable`. In this case, the parser must be able to figure out what kind of metavariable it is. It is also possible to specify the type of a metavariable by enclosing the type in :'s, concatenated directly to the metavariable name.

Some examples of semantic matches that can be given as an argument to `-sp` are as follows:

- `f(e)`: This only matches the expression `f(e)`.
- `f(E)`: This matches a call to f with any argument.
- `F(E)`: This gives a parse error; the semantic patch parser cannot figure out what kind of metavariable `F` is.
- `F:identifier:(E)`: This matches any one argument function call.
- `f:identifier:(e:struct foo *:)`: This matches any one argument function call where the argument has type `struct foo *`. Since the types of the metavariables are specified, it is not necessary for the metavariable names to begin with a capital letter.

- `F:identifier:(F)`: This matches any one argument function call where the argument is the name of the function itself. This example shows that it is not necessary to repeat the metavariable type name.

- `F:identifier:(F:identifier:)`: This matches any one argument function call where the argument is the name of the function itself. This example shows that it is possible to repeat the metavariable type name.

`When` constraints, *e.g.* `when != e`, are allowed but the expression `e` must be represented as a single token. The generated semantic match behaves as though there were a `*` in front of every token.

9.14 Iteration

It is possible to iterate Coccinelle, giving the subsequent iterations a different set of virtual rules or virtual identifier bindings. Coccinelle currently supports iteration with both OCaml and Python scripting. An example with OCaml is fond in `demos/iteration.cocci`, a Python example is found in `demos/python_iteration.cocci`.

The OCaml scripting iteration example starts as follows.

```
virtual after_start

@initialize:ocaml@

let tbl = Hashtbl.create(100)

let add_if_not_present from f file =
try let _ = Hashtbl.find tbl (f,file) in ()
with Not_found ->
   Hashtbl.add tbl (f,file) file;
   let it = new iteration() in
   (match file with
     Some fl -> it#set_files [fl]
   | None -> ());
   it#add_virtual_rule After_start;
   it#add_virtual_identifier Err_ptr_function f;
   it#register()
```

The respective Python scripting iteration example starts as follows:

```
virtual after_start

@initialize:python@
@@

seen = set()

def add_if_not_present (source, f, file):
    if (f, file) not in seen:
        seen.add((f, file))
        it = Iteration()
        if file != None:
            it.set_files([file])
        it.add_virtual_rule(after_start)
        it.add_virtual_identifier(err_ptr_function, f)
        it.register()
```

The virtual rule `after_start` is used to distinguish between the first iteration (in which it is not considered to have matched) and all others. This is done by not mentioning `after_start` in the command line, but adding it on each iteration.

The main code for performing the iteration is found in the function `add_if_not_present`, between the lines calling `new iteration` and `register`. `New iteration` creates a structure representing the new iteration. `set_files` sets the list of files to be considered on the new iteration. If this function is not called, the new iteration treats the same files as the current iteration. `add_virtual_rule a` has the same effect as putting `-D a` on the command line. If using OCaml scripting instead of Python scripting the first letter of the rule name is capitalized, although this is not done elsewhere (technically, the rule name is an OCaml constructor). `add_virtual_identifier x v` has the same effect as putting `-D x=v` on the command line. Again, when using OCaml scripting there is a case change. `extend_virtual_identifiers()` (not shown) preserves all virtual identifiers of the current iteration that are not overridden by calls to `add_virtual_identifier`. Finally, the call to `register` queues the collected information to trigger a new iteration at some time in the future.

Modification is not allowed when using iteration. Thus, it is required to use the `-no-show-diff`, unless the semantic patch contains `*`s (a semantic match rather than a semantic patch).

When using Python scripting a tuple may be used to ensure that the same information is not enqueued more than once. When using OCaml scripting a hash table may be used for the same purpose. Coccinelle itself provides no support for obtaining information about what work has been queued and as such addressing this with scripting is necessary.

9.15 .cocciconfig support

Coccinelle supports enabling custom options to be preferred when running spatch. This is supported through the search of .cocciconfig files in each of the following directories, later lines extend and may override earlier ones:

- Your current user's home directory is processed first
- Your directory from which spatch is called is processed next
- The directory provided with the –dir option is processed last, if used

Newlines, even with are not tolerated in attribute values. An example follows:

```
[spatch]
options = --jobs 4
options = --show-trying
```

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 the changes between the original and transformed code, using the program `diff`. `--sp-file` is optional in this and the following cases.
- `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`.

The last four commands above produce a patch describing any changes. This patch can typically be applied to the source code using the command `patch -p1`, like any other patch. Alternatively, the option `--in-place` both produces the patch and transforms the code in place.

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.

⬙ **--debug-parse-cocci** Print some information about the definition of virtual rules and the bindings of virtual identifiers. This is particularly useful when using iteration, as it prints out this information for each iteration.

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. These options are optional. If they are not used, the single file whose name ends in `.cocci` is assumed 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.

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.

◇ **--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.

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`.

◇ **--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.

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.

◆ **--use-idutils** [⟨**file**⟩] 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.

The optional file name option is the name of the file in which to find the index. It has been reported that the viewer seascope can be used to generate an appropriate index. If no file name is specified, the default is .id-utils.index. If the filename is a relative path name, that path is intereted relative to the target directory. If the filename is an absolute path name, beginning with /, it is used as is.

◆ **--use-coccigrep** Use a version of grep implemented in Coccinelle to check that selected files are relevant to the semantic patch. This option is only relevant to the case of working on a complete directory, when parallelism is requested (max and index options). Otherwise it is the default, except when multiple files are requested to be treated as a single unit. In that case grep is used.

Note that coccigrep or grep is used even if glimpse or id-utils is selected, to account for imprecision in the index (glimpse at least does not distinguish between underline and space, leading to false positives).

◇ **--selected-only** Just show what files will be selected for processing.

◆ **--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.

◇ **--file-groups** Specify a file that contains the list of files to process. Files should be listed one per line. Blank lines should be used to separate the files into *groups*. All files within a single group will be treated at once. This is useful, for example, if one wants to process a complete driver, that consists of more than one file, and it is necessary to consider the interaction between code fragments that are present in the different files. Single-line comments beginning with `//` can be used freely and are ignored.

--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`". The result is put in the file `/tmp/file.res`. If writing a file in /tmp with a non-fresh name is a concern, then do not use this option.

--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. If the argument -()-expected-score-file is provided, then that file is used for the result. Otherwise, the result goes in "tests/SCORE_expected.sexp". **Warning:** It is intended that not all of the test cases provided with Coccinelle actually pass.

◇ **--test-spacing** Like --testall, but ensures that the spacing is the same as in the .res file. If the argument -()-expected-spacing-score-file is provided, then that file is used for the result. Otherwise, the result goes in "tests/SCORE_spacing_expected.sexp".

--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.

◇ **--verbose-parsing** Show parsing errors in the C file, as well as information about attempts to accommodate such errors. This implicitly sets **--parse-error-msg.**

◇ **--parse-handler ⟨file⟩** Loads the file containing the OCaml code in charge of parse error reporting. This function should have arguments 1) the line number containing the error, 2) the sequence of tokens, the starting and ending line of the function containing the error, and array containing the lines of the file containing the error, and the pass of the parser on which the error occurs. This function should then be passed to the function `Parse_c.set_parse_error_function`.

◇ **--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.

◇ **--noif0-passing** 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.

◇ **--defined** s The string s is a comma-separated list of constants that should be considered to be defined, with respect to uses of `#ifdef` and `#ifndef` in C code. No spaces should appear in s. Multiple **--defined** arguments can be provided and the list of strings accumulates. For the provided strings any `#else`s of `#ifdef`s are ignored and any `#ifndef`s are ignored, unless the argument **--noif0-passing** is also given, in which case **--defined** has no effect. Note that occurrences of `#define` in the C code have no effect on the list of defined constants.

This option now applies also to `#if` in which case the string has be exactly as it appears in the code, minus any leading whitespace or tabs, and minus any comments. Not that there is currently no way to provide information about the expressions used in `#elif`.

◇ **--undefined** s Analogous to **--defined** except that the strings represent constants that should be considered to be undefined.

--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 reachable by the specified path from the directory of the .c file. In this case, for non-local includes, specified with <>, Coccinelle will also search from the directories specified with `-I` for `.h` files with the same name as the `.c` file. **--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 to specify multipls include paths.

◆ **--include-headers-for-types** Header files are parsed to collect type information, but are not involved in the subsequent matching and transformation process.

◇ **--include ⟨file⟩** This option give the name of a file to consider as being included in each processed file. The file is added to the end of the file's list of included files. The complete path name should be given; the **-I** options are not taken into account to find the file. This option can be used several times to include multiple files.

◇ **--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.

◇ **--ibm** Make a effort to parse IBM C code. Currently decimal declarations are supported.

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

◆ **-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.

◇ **--reverse** Inverts the semantic patch before applying it. A potential use case is backporting changes to previous versions. If a semantic patch represents an API change, then the reverse undoes the API change. Note that inverting a semantic patch is not always possible. In particular, the composition of a semantic patch with its inverse is not guaranteed to be an empty patch.

10.5 Generation of the result

Normally, the only output is the differences between the original code and the transformed code obtained using the program `diff` with the unified format option. If stars are used in column 0 rather than − and +, then the − lines in the output 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.

⬙ **--indent** n The number of spaces to indent, if no other information is available. If this information is not provided, then the default indentation is a tab. This option is thus particularly relevant to projects that don't use tabs.

◇ **-o ⟨file⟩** This causes the transformed code to be placed in the file `file`. The difference between the original code and the transformed code is still printed to the standard output using `diff` with the unified format option. This option only makes sense when − and + are used.

◇ **--in-place** Modify the input file to contain the transformed code. The difference between the original code and the transformed code is still printed to the standard output using `diff` with the unified format option. 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** s The suffix s 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. The difference between the original code and the transformed code is still printed to the standard output using `diff` with the unified format option. This option only makes sense when − and + are used.

◇ **--no-show-diff** Normally, the difference between the original and transformed code is printed on the standard output. This option causes this not to be done.

◇ **-U** Set number of context lines to be provided by `diff`.

◇ **--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.

10.6 Other options

10.6.1 Version information

⬙ **--version** The version of Coccinelle is printed on the standard output. No other options are allowed.

⬙ **--date** The date of the current version of Coccinelle are printed on the standard output. 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. A timeout of 0 is no timeout.

◇ **--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.

--profile-per-file Like **--profile**, but generates information after processing each file.

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

10.6.4 Parallelism

⬘ **--jobs** ⟨**int**⟩ Run the specified number of jobs in parallel. Can be abbreviated as **-j**. This option is not compatible with the use of a `finalize` rule in the semantic patch, as there is no shared memory and the effect of a `finalize` rule is thus not likely to be useful. This option furthermore creates a temporary directory in the directory from which spatch is executed that has the name of the semantic patch (without its extension) and that contains stdout and stderr files generated by the various processes. When the semantic patch completes, the contents of these files are printed to standard output and standard error, respectively, and the directory is removed.

⬘ **--tmp-dir** ⟨**string**⟩ Specify the name of the temporary directory used to hold the results obtained on the different cores with the **-j** option.

--chunksize ⟨**int**⟩ The specified number of files are dispatched as a single unit of parallelism. This option is only interesting with the options **--all-includes** or **--recursive-includes**, when combined with the option **--include-headers-for-types**. In this case, parsed header files are cached. It is only the files that are treated within a single chunk that can benefit from this cache, due to the lack of shared memory in ocaml.

10.6.5 External analyses

--external-analysis-file Loads in the contents of a database produced by some external analysis tool. Each entry contains the analysis result of a particular source location. Currently, such a database is a .csv file providing integer bounds or an integer set for some subset of the source locations that references an integer memory location. This database can be inspected with coccilib functions, e.g. to control the pattern match process.

10.6.6 Miscellaneous

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

--pad, --xxx, --l1

Part III

Appendix

GNU Free Documentation License

GNU Free Documentation License Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.