The SmPL Grammar (version 0.2.3)

Research group on Coccinelle

March 20, 2011

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

1 Program

program	::=	*include_cocci** *changeset*$^+$
include_cocci	::=	`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 string or 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.

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 ::= *pnrule*
| *dep* `&&` *dep*
| *dep* `||` *dep*

pnrule ::= id
| `!` id
| `ever` id
| `never` id
| `(`*dep*`)`

iso ::= `using` *string* (`,` *string*)*

disable-iso ::= `disable` *COMMA_LIST*(id)

exists ::= `exists`
| `forall`

COMMA_LIST(*elem*) ::= *elem* (`,` *elem*)*

The keyword `disable` is normally used with the names of isomorphisms defined in standard.iso or whatever isomorphism file has been included. There are, however, some other isomorphisms that are built into the implementation of Coccinelle and that can be disabled as well. Their names are given below. In each case, the text 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 is it the qualifier (i.e., const or volatile) that does not have to be specified in the SmPL code, but may be present in the C code.
- `value_format`: Integers in various formats, e.g., 1 and 0x1, are considered to be equivalent in the matching process.
- `comm_assoc`: An expression of the form *exp bin_op* `...`, where *bin_op* is commutative and associative, is considered to match any top-level sequence of *bin_op* operators containing *exp* as the top-level argument.

The possible types of metavariable declarations are defined by the grammar rule below. Metavariables should occur at least once in the transformation immediately following their declaration. Fresh 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.

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* `;`
| `type` *ids* `;`
| `statement` [`list`] *ids* `;`
| `declaration` [`list`] *ids* `;`
| `field` [`list`] *ids* `;`
| `typedef` *ids* `;`
| `declarer name` *ids* `;`
| `declarer` *COMMA_LIST*(*pmid_with_regexp*) `;`
| `declarer` *COMMA_LIST*(*pmid_with_not_eq*) `;`
| `iterator name` *ids* `;`
| `iterator` *COMMA_LIST*(*pmid_with_regexp*) `;`
| `iterator` *COMMA_LIST*(*pmid_with_not_eq*) `;`
| [`local`] `idexpression` [*ctype*] *COMMA_LIST*(*pmid_with_not_eq*) `;`
| [`local`] `idexpression` [{*ctypes*} $\star^*$] *COMMA_LIST*(*pmid_with_not_eq*) `;`
| [`local`] `idexpression` $\star^+$ *COMMA_LIST*(*pmid_with_not_eq*) `;`
| `expression list` *ids* `;`
| `expression` $\star^+$ *COMMA_LIST*(*pmid_with_not_eq*) `;`
| `expression enum` $\star^*$ *COMMA_LIST*(*pmid_with_not_eq*) `;`
| `expression struct` $\star^*$ *COMMA_LIST*(*pmid_with_not_eq*) `;`
| `expression union` $\star^*$ *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*} $\star^*$ *COMMA_LIST*(*pmid_with_not_ceq*) `;`
| {*ctypes*} $\star^*$ `[ ]` *COMMA_LIST*(*pmid_with_not_eq*) `;`
| `constant` [*ctype*] *COMMA_LIST*(*pmid_with_not_eq*) `;`
| `constant` [{*ctypes*} $\star^*$] *COMMA_LIST*(*pmid_with_not_eq*) `;`
| `position` [`any`] *COMMA_LIST*(*pmid_with_not_eq_mid*) `;`

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 ideexpression.

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 gives to macros in Linux usually have this form.

An identifier is the name of a structure field, a macro, a function, or a variable. Is 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.

```
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 [!= mid]
                             | pmid [!= { COMMA_LIST(mid) }]
```

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

`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 8.

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`.

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] @
                       | @ finalize:language [depends on dep] @
language             ::= python
                       | ocaml
script_metadecl      ::= id << rulename_id . id ;
                       | 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.

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.

4 Transformation

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

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

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

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

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

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

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

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

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

Lines may be annotated with an element of the set $\{-, +, \star\}$ or the singleton `?`, or one of each set. `?` represents at most one match of the given pattern. $\star$ 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 with any marking.

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

5 Types

ctypes ::= *COMMA_LIST*(*ctype*)

ctype ::= [*const_vol*] *generic_ctype* `*`*
| [*const_vol*] `void *`$^+$
| `(`*ctype* `(|` *ctype*$)^*$`)`

const_vol ::= `const`
| `volatile`

generic_ctype ::= *ctype_qualif*
| [*ctype_qualif*] `char`
| [*ctype_qualif*] `short`
| [*ctype_qualif*] `int`
| [*ctype_qualif*] `long`
| [*ctype_qualif*] `long long`
| `double`
| `float`
| `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.

6 Function declarations

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

7 Declarations

decl_var ::= *common_decl*
| [*storage*] *ctype* *COMMA_LIST*(*d_ident*) `;`
| [*storage*] [*const_vol*] id *COMMA_LIST*(*d_ident*) `;`
| [*storage*] *fn_ctype* `(` `*` *d_ident* `)` `(` *PARAMSEQ*(*name_opt_decl*, ε) `)` `=` *initialize* `;`
| `typedef` *ctype* *typedef_ident* `;`

one_decl ::= *common_decl*
| [*storage*] *ctype* *id*`;`
| [*storage*] [*const_vol*] id *d_ident* `;`

common_decl ::= *ctype*`;`
| *funproto*
| [*storage*] *ctype* *d_ident* `=` *initialize* `;`
| [*storage*] [*const_vol*] id *d_ident* `=` *initialize* `;`
| [*storage*] *fn_ctype* `(` `*` *d_ident* `)` `(` *PARAMSEQ*(*name_opt_decl*, ε) `)` `;`
| *decl_ident* `(` [*COMMA_LIST*(*expr*)] `)` `;`

initialize ::= *dot_expr*
| metaid$^{\text{Initialiser}}$
| `{` [*COMMA_LIST*(*dot_expr*)] `}`

init_list_elem ::= *dot_expr*
| *designator* `=` *dot_expr*
| *id* `:` *dot_expr*

designator ::= `.` *id*
| `[` *dot_expr* `]`
| `[` *dot_expr* `...` *dot_expr* `]`

decl_ident ::= DeclarerId
| metaid$^{\text{Declarer}}$

8 Statements

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

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

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

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

9 Expressions

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

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

10 Constants, Identifiers and Types for Transformations

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

11 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.

11.1 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();
}
```

11.2 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);
}
```

11.3 Introduction of a macro

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

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

Semantic patch to introduce uses of the `DIV_ROUND_UP` macro

```
@ haskernel @
@@

#include <linux/kernel.h>

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

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

Example of a generated patch hunk

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

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

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

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

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

@ disable unlikely @
expression E,f;
@@

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

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

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

11.4 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;
```

11.5 Reference counter: the of_xxx API

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

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

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

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

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

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

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

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

```
1 * 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]]
2 [[view:/linux-next/arch/powerpc/platforms/pseries/setup.c::face=ovl-face2::
    linb=250::colb=3::cole=9][return]]
3 * 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]]
4 [[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.

```
1 @ script:python @
2 p1 << r.p1;
3 p2 << r.p2;
4 @@
5
6 cocci.print_main(p1)
7 cocci.print_sec(p2,"return")
```

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

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

11.6 Filtering identifiers, declarers or iterators with regular expression

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

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

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

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

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

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

t foo () {...}

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

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

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

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

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

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

12 Tips and Tricks

12.1 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.