The SmPL Grammar (version 1.0.8)

Research group on Coccinelle

September 2, 2020

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`.

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 used 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.

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` [*scope*] *dep*] [*iso*] [*disable-iso*] [*exists*] [*rulekind*]

scope ::= exists
| forall

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`

rulekind ::= `expression`
| `identifier`
| `type`

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 also similar to `optional_storage`, except that here it is an attribute (e.g., __init) 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.
- `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 of the directory to which Coccinelle was applied. In the latter case, the string is matched against the complete pathname. A trailing `/` is added to the specified subdirectory name, to ensure that a complete subdirectory name is matched. 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.

As metavariables are bound and inherited across rules, a tree of environments is built up. A rule is processed only once for all of the branches that have the same metavariable bindings for the set of variables that the rule depends on. Different branches, however, may be derived from the success or failure of different sets of rules. A `depends on` clause can further indicate whether the clause should be satisfied for all the branches (`forall`) or only for one (`exists`). `exists` is the default. These annotations can for example be useful when one rule binds a metavariable `x`, subsequent rules have the effect of testing good and bad properties of `x`, and a final rule may want to ensure that all occurrences of `x` have the good property (`forall`) or none have the bad property (`exists`). `forall` and `exists` are currently only supported at top level, not under conjunction and disjunction.

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 ::= `fresh identifier` *pmids_with_seed* `;`
| `metavariable` *pmids_with_constraints* `;`
| `identifier` *pmvids_with_constraints* `;`
| `identifier` *list* *pmvids_with_constraints* `;`
| `field` [*list*] *pmids_with_constraints* `;`
| `parameter` [*list*] *pmids_with_constraints* `;`
| `type` *pmids_with_constraints* `;`
| `statement` [`list`] *pmids_with_constraints* `;`
| `declaration` *pmids_with_constraints* `;`
| `initialiser` [*list*] *pmids_with_constraints* `;`
| `initializer` [*list*] *pmids_with_constraints* `;`
| [`local` | `global`] `idexpression` [*ctype*] *pmids_with_constraints* `;`
| [`local` | `global`] `idexpression` [`{` *ctypes* `}` $\star^*$] *pmids_with_constraints* `;`
| [`local` | `global`] `idexpression` $\star^+$ *pmids_with_constraints* `;`
| `expression` *list* *pmids_with_constraints* `;`
| `expression` [`enum` | `struct` | `union`] $\star^*$ *pmids_with_constraints* `;`
| *ctype* [`[ ]`] *pmids_with_constraints* `;`
| `{` *ctypes* `}` $\star^*$ [`[ ]`] *pmids_with_constraints* `;`
| `constant` [*ctype*] *pmids_with_constraints* `;`
| `constant` [`{` *ctypes* `}` $\star^*$] *pmids_with_constraints* `;`
| `format` [*list*] *pmids_with_constraints*`;`
| `assignment operator` *COMMA_LIST*(*assignopdecl*) `;`
| `binary operator` *COMMA_LIST*(*binopdecl*) `;`
| `unary operator` *COMMA_LIST*(*unopdecl*) `;`
| `position` [`any`] *pmids_with_constraints* `;`
| `symbol` *pmids*`;`
| `typedef` *pmids* `;`
| `attribute name` *ids* `;`
| `declarer name` *ids* `;`
| `declarer` *pmids_with_constraints* `;`
| `iterator name` *ids* `;`
| `iterator` *pmids_with_constraints* `;`

list ::= `list`
| `list [` id `]`
| `list [` integer `]`

assignopdecl ::= *pmid* [= *assignop_constraint*]

assignop_constraint ::= {*COMMA_LIST*(*assign_op*)}
| *assign_op*

binopdecl ::= *pmid* [= *binop_constraint*]

binop_constraint ::= {*COMMA_LIST*(*bin_op*)}
| *bin_op*

unopdecl ::= *pmid* [= *unop_constraint*]

unop_constraint ::= {*COMMA_LIST*(*unary_op*)}
| *unary_op*

fresh identifier metavariables can only be used in + code and will generate new identifiers according to the optionally given seed:

- if none is given, then one will be requested on the command line during execution of the semantic patch
- if a single string is given then, that string will be suffixed by an increasing number, ensuring that spatch does not use the same identifier in multiple instances of a rule or in between rules

- if a concatenation of strings and/or ids is provided using the `##` operator, or a single id is given, then the strings will be kept as is and each id will be replaced by its corresponding content (as string) for each evaluation of the rule
- if a script is given, then it must return a string and the result will be used as is

Examples are found in `demos/plusplus1.cocci` and `demos/plusplus2.cocci`

`metavariable` declares a metavariable for which the parser tries 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 seem 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.

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.

The **list** modifier allows to match over multiple elements of a given kind in a row and store them as one metavariable. It is possible to specify its length. If no length element is provided then the list will be the longest possible. If an integer length is provided, then only lists of the given length are matched. If an id is provided, then it will store the length of the matched list. This id can be used to ensure other lists have the same length, or can be manipulated in script code.

An **identifier list** is only used for the parameter list of a macro. It matches multiple identifiers in a row and stores them as one metavariable.

A **field** only matches an identifier that is a structure field.

A **parameter** matches a parameter declaration. Arguments (values given at function call) are not matched through this but using other kinds of metavariables (e.g. **expression**).

A **type** matches a type appearing in code whether it is in the declaration of a function, a variable, in a cast or anywhere else where it is explicitly a type. It also matches a type name defined by a **typedef**

A **statement** matches anything that falls into the statement definition of the C99 standard.

A **statement list** can only match a complete sequence of statements between braces. Therefore, no size can be specified for it and no statement can contiguously surround it for context (it has to be absorbed).

A **declaration** matches the declaration of one or more variables sharing the same type specification.

An **initialiser** or **initializer** matches the right hand side of a declaration.

An **idexpression** is a variable used as an expression. It is useful to restrict a match to be both an identifier and to have a particular type. A more complex description of a location, such as a->b is considered to be an **expression** not an **idexpression**. The optional **local** modifier restricts the matched variable to be a local variable. The optional **global** indicates that the matched variable is not a local one. If neither **local** or **global** is specified, then any variable reference can be matched. It is possible to specify a *ctype* or a set of them and/or a pointer level using $\star$ to restrict the types of variables that can be matched.

An **expression** is any piece of code that falls into the expression definition of the C99 standard. Therefore, any combination of sequences of operators and operands that computes a value, designates an object or a function, or generates side effects is matched as en expression. It is possible to specify some type information using **enum**, **struct**, or **union**, and/or a pointer level using $\star$ to restrict the types of expressions that can be matched. It is possible to only match expressions of a specific *ctype* or a set of them with a pointer level using $\star$ by writing these instead of the **expression** designator pattern. One can also specify the matched expression must be of array type by adding brackets after the initial type specification. The *ctype* and *ctypes* nonterminals are used by both the grammar of metavariable declarations and the grammar of transformations, and are defined on page 17.

A **constant** metavariabe matches a constant in the code, such as 27. It also considers an uppercase identifier as a constant as well, because the names given to macros in Linux usually have this form.

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`.

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.

An **assignment operator** (resp. **binary operator**) metavariable matches 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.

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.

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.

A **symbol** declaration specifies that the provided identifiers should be considered to be 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, furthermore, 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.

A **typedef** declaration specifies that the provided identifiers should be considered as types when encountered in the code for match. Such a declaration is useful to ensure spatch will match some identifiers as types properly when the declaration is not available in the processed code. It is not always necessary to specify a type that has no declaration in the given code is a type, because spatch can sometimes extrapolate that information from context. 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, furthermore, 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.

An **attribute name** declaration indicates the given identifiers should be considered to be attributes.

A **declarer** is a macro call used at top level which generates a declaration. Such macros are used in the Linux kernel.

The **name** modifier specifies that instead of declaring a metavariable to match over some kind, the identifiers are to be considered as elements of that kind when they appear in the code.

An **iterator** is a macro call used in place of an iteration statement header (e.g. `for (size_t i = 0; i < 10; ++i)`) which generates it. Such macros are used in the Linux kernel.

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.

ids ::= *COMMA_LIST*(id)
pmids ::= *COMMA_LIST*(*pmid*)
pmids_with_constraints ::= *COMMA_LIST*(*pmid* [*constraints*])
pmvids_with_constraints ::= *COMMA_LIST*(*pmvid* [*constraints*])
pmids_with_seed ::= *COMMA_LIST*(*pmid* [*seed*])
pmvid ::= *pmid*
| `virtual.`id
pmid ::= id
| *mid*
mid ::= rulename_id`.`id
constraints ::= *ANDAND_LIST*(*constraint*)
constraint ::= *compare_constraint*
| *regexp_constraint*
| `:` *script*
compare_constraint ::= *id_compare_constraint*
| *int_compare_constraint*
id_compare_constraint ::= `=` *pmid*
| `=` `{` *COMMA_LIST*(*pmid*) `}`
| `!=` *pmid*
| `!=` `{` *COMMA_LIST*(*pmid*) `}`
int_compare_constraint ::= `=` integer
| `=` `{` *COMMA_LIST*(integer) `}`
| `!=` integer
| `!=` `{` *COMMA_LIST*(integer) `}`
regexp_constraint ::= `=~` *regexp*
| `!~` *regexp*
seed ::= `=` *string*
| `=` *CONCAT_LIST*(*string* | *pmid*)
| `=` *script*
script ::= `script:ocaml (` *COMMA_LIST*(*mid*) `) {` *expr* `}`
| `script:python (` *COMMA_LIST*(*mid*) `) {` *expr* `}`
ANDAND_LIST(X) ::= X [`&&` *ANDAND_LIST*(X)]
CONCAT_LIST(X) ::= X [`##` *CONCAT_LIST*(X)]

A meta identifier 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`.

Most metavariables can be given constraints to indicate authorized/forbidden values. These constraints fall in different categories:

- comparison constraints to indicate that a metavariable must be equal to or different from some integer values or some other metavariables
- regexp constraints to indicate that a metavariable's matched code must satisfy or must not satisfy the given regular expression
- script constraints to indicate that the metavariable must validate some arbitrary constraint written in a script language. A script constraint must return a boolean value

Multiple constraints can be attached to a single metavariable by separating them using `&&`, and all the constraints must be met at the same time for their composition to be true. It is also possible to include inherited identifier metavariables among the constraints.

Metavariables can be associated with constraints implemented as OCaml or python script code. The form of the code is somewhat restricted, due to the fact that it passes through the Coccinelle semantic patch lexer, before being converted back to a string to be passed to the scripting language interpreter. It is thus best to avoid complicated code in the constraint itself, and instead to define relevant functions in an `initialize` rule. The code must represent an expression that has type bool in the scripting language. The script code can be parameterized by any inherited metavariables. It is implicitly parameterized by the metavariable being declared. In the script, the inherited metavariable parameters are referred to by their variable names, without the associated rule name. The script code can also be parameterized by metavariables defined previously in the same rule. Such metavariables must always all be mentioned in the same "rule elem" as the metavariable to which the constraint applies. Such a rule elem must also not contain disjunctions, after disjunction lifting. The result of disjunction lifting can be observed using `--parse-cocci`. A rule elem is eg an atomic statement, such as a return or an assignment, or a loop header, if header, etc. The variable being declared can also be referenced in the script code by its name. All parameters, except position variables, have their string representation. An example is in `demos/poscon.cocci`.

Script constraints may be executed more than once for a given metavariable binding. Executing the script constraint does not guarantee that the complete match will work out; the constraints are executed within the matching process.

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.

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. A script is only run if all metavariables are bound, either by inheritance or by a default value given with =.

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. 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.

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.

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();
```

Note that the region matched by "..." can be empty.

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 to be matched 0 or more times, i.e., it 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();
```

4.3 An example

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

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

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.c has a linear control flow, i.e., 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).

```
1 pmccabe /flow1.c
2 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.

```
1 pmccabe flow2.c
2 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.

5 Transformation

Coccinelle semantic patches are able to transform C code.

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. The shortest path constraint is implemented by requiring that the pattern (if any) appearing immediately before the dots and the pattern (if any) appearing immediately after the dots are not matched by the code matched by the dots. `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. The code is presented with lines containing a match of a starred line preceded by $-$, but this is not intended as a removal and applying the output as a patch to the original code will likely not result in correct 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__((...))`).

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.

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`
| `long double complex`
| `double complex`
| `float complex`
| `size_t`
| `ssize_t`
| `ptrdiff_t`
| `enum` *id* `{` *PARAMSEQ*(*dot_expr*`,` *exp_whencode*) [`,`] `}`
| [`struct`| `union`] id [`{` *struct_decl_list*`*` `}`]
| `typeof` `(` *exp* `)`
| `typeof` `(` *ctype* `)`
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.

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
| metaid$^{\text{Id}}$
| *OR*(*funid*)
param ::= *type* id
| metaid$^{\text{Param}}$
| metaid$^{\text{ParamList}}$
| `.....`
decl ::= *ctype* *id*
| *fn_ctype* `(*` *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 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.

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* *COMMA_LIST*(*typedef_ident*) `;`
| `typedef` *ctype* *typedef_ident* `[` *expr* `]` `;`
| `typedef` *ctype* *typedef_ident* `[` *expr* `]` `[` *expr* `]` `;`
| *OR*(*decl_var*)
| *AND*(*decl_var*)

one_decl ::= *common_decl*
| [*storage*] *ctype* *id* [*attribute*] `;`
| *OR*(*one_decl*)
| *AND*(*one_decl*)
| [*storage*] [*const_vol*] id *d_ident* `;`

common_decl ::= *ctype* `;`
| *funproto*
| [*storage*] *ctype* *d_ident* [*attribute*] `=` *initialize* `;`
| [*storage*] [*const_vol*] id *d_ident* [*attribute*] `=` *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.

An attribute begins with __ or is declared as an `attribute name` in the semantic patch. In practice, only one attribute is currently allowed after the variable name in a variable declaration.

Coccinelle supports declaring multiple variables or structure fields in the C code, but not in the SmPL code. It is possible to remove a variable from within a declaration of multiple variables with a pattern that removes a complete single-variable declaration, e.g., `- int x;`. The type and the semicolon are only removed if all of the variables are removed. It is also possible to specify to entirely remove such a declaration and replace it with something else. The replacement of a declaration only matches if the addition is done with `++`, allowing multiple additions. This is also only allowed if there is no implicitly matched information on the type, such as `extern` or `static`. When the transformation cannot be made, there is no crash, simply a match failure. A message is given for this with the `--debug option`.

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 ...
single_stmt   ::= stmt
                | OR(stmt)
                | AND(stmt)
decl_stmt     ::= metaid^StmtList
                | decl_var
                | stmt
                | expr
                | OR(stmt_seq)
                | AND(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)* )
AND(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.

OR(*stmt_seq*) and *AND*(*stmt_seq*) must have something other than an expression in the first branch. If an expression appears there, they are parsed as their *expr* counterparts, *i.e.*, all branches must be expressions.

All matching done by a SmPL rule is done intraprocedurally. Thus "..." does not extend from one function to the next one in the same file and it does not extend from one function over a function call into the called function.

`#pragma` C code can only be matched against when the entire pragma is on one line in the C code. The use of continuation lines, via a backslash, will cause the matching to fail.

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^IdExp
| metaid^Const
| *const*
| `(`*dot_expr*`)`
| *OR*(*exp*)
| *AND*(*exp*)
arg ::= *nest_expr*
| metaid^ExpList

exp_whencode ::= `when != ` *expr*
assign_op ::= `= | -= | += | *= | /= | %=`
| `&= | |= | ^= | <<= | >>=`
bin_op ::= `* | / | % | + | -`
| `<<| >>| ^ | & | |`
| `< | > | <= | >= | == | != | && | ||`
unary_op ::= `++ | - | & | * | + | - | !`

11 Constants, Identifiers and Types for Transformations

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

Conjunctions for identifiers are, as indicated by the BNF, not currently supported.

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`

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.

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 found 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 $\star$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.

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
```

16 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.

16.1 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();
}
```

16.2 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);
}
```

16.3 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__)
```

16.4 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;
```

16.5 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 script rule depends only 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.

```
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 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.

16.6 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
```

17 Tips and Tricks

This section presents some tips and tricks for using Coccinelle.

17.1 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.