INTRODUCTION TO COCCINELLE AND SMPL

Linuxcon Japan, 2016

Vaishali Thakkar

(vaishali.thakkar@oracle.com)
Prerequisites

- Source code of the Linux kernel version 4.6
- Latest version of the Coccinelle
  - Either install it from the package manager [Coccinelle is available with around 10 linux distros including Fedora, Ubuntu, Debian, ArchLinux etc.].
  - Or build it from the source. (https://github.com/coccinelle/coccinelle)
Code Maintenance Issues

- **Software evolution:**
  - Refactoring code to use newer APIs

  ```c
  init_timer(&cf->timer);
  cf->timer.function = omap_cf_timer;
  cf->timer.data = (unsigned long) cf;
  setup_timer(cf->timer, omap_cf_timer, (unsigned long)cf);
  ```

  - Need to find all parts of the code that need updating
  - Process should be fast, reliable and systematic
  - However, things are never straightforward
Code Maintenance Issues

• **Software evolution:**
  - Refactoring code to use newer APIs
  - Need to find all parts of the code that need updating
  - Process should be fast, reliable and systematic
  - However, things are never straightforward

• **Software robustness:**
  - Are the programmers following the standards?
  - Is the code accounting for all errors that can take place?
  - Is the written code overly defensive?
Code Maintenance Issues

- **Software evolution:**
  - Refactoring code to use newer APIs
  - Need to find all parts of the code that need updating
  - Process should be fast, reliable and systematic
  - However, things are never straightforward

- **Software robustness:**
  - Are the programmers following the standards?
  - Is the code accounting for all errors that can take place?
  - Is the written code overly defensive?

- **The Human Factor:**
  - Mistakes can always happen
Coccinelle

- Program matching and transformation tool
- Independent of the compilation process
- Very intuitive patch like style
- Used by several communities:
  - Linux Kernel: 5K+ patches
  - QEMU: 200+ patches
  - systemd: 80+ patches
Semantic Patch Language (SmPL)

- Abstract C-like grammar
- Independent of the compilation process
- Metavariables are used to abstract over sub-terms in code
  - If an expression matches within a pattern, it can be tracked throughout its presence in the code e.g. variable names, typedefs
- “...” is used to abstract over code sequences
  - Used as don’t care
  - Variants are used as syntactic sugar for + and ? in regular expressions
- Lines can be annotated with {-,+,*}
  - Transformations are described using patch-like style (-/+)
  - Matching employs *
Example: Using BIT macro

- Bit masking is preferrably done using the BIT macro

```c
-BUILD_BUG_ON(max >= (1 << 16));
+BUILD_BUG_ON(max >= (BIT(16)));```

Example: Using BIT macro

- Bit masking is preferrably done using the BIT macro

```c
- BUILD_BUG_ON(max >= (1 << 16));
+ BUILD_BUG_ON(max >= (BIT(16)));
```

- Code we should focus on for building a semantic patch:

```c
- 1 << 16
+ BIT(16)
```
Example: Using BIT macro

- Bit masking is preferrably done using BIT macro

- BUILD_BUG_ON(max >= (1 << 16));
+ BUILD_BUG_ON(max >= (BIT(16)));

- Code we should focus on for building a semantic patch:

- 1 << 16
+ BIT(16)

- Is 16 important here?
Example: Using BIT macro (Contd.)

- Do we care about number of shifts?

- if (opts & (1 << REISERFS_LARGETAIL))
+ if (opts & (BIT(REISERFS_LARGETAIL)))
Example: Using BIT macro (Contd.)

- Do we care about number of shifts?

```c
- if (opts & (1 << REISERFS_LARGETAIL))
+ if (opts & (BIT(REISERFS_LARGETAIL)))
```

- Use metavariables

```c
constant c;
@@
-1 << c
+BIT(c)
```
Example: Using BIT macro (Contd.)

- Constant will capture numbers and defined constants

- What if we had something like

  \[ 1 \ll (31 - \text{inode->i_sb->s_blocksize_bits}) \]
Example: Using BIT macro (Contd.)

- Constant will capture numbers and defined constants

- What if we had something like

  \[ 1 \ll (31 - \text{inode->i_sb->s_blocksize_bits}) \]

- expression to the rescue

  ```
  @@
  expression E;
  @@
  
  -1 \ll E
  +BIT(E)
  ```
Metavariabes

Example: $x \rightarrow y = m \rightarrow n + 1$;

- **Constant**: match patterns on values and constants
  
  e.g. numbers like 2,3 and defined constants in a code
Metavariables

Example: \( x \rightarrow y = m \rightarrow n + 1; \)

- **Constant**: match patterns on values and constants
e.g. numbers like 2, 3 and defined constants in a code

- **Expression**: match patterns on constants and complex subterms
e.g. `struct->elem`, `x-y`, `func(arg)` etc
Metavariables

Example: \(x \rightarrow y = m \rightarrow n + 1;\)

- **Constant**: match patterns on values and constants
  e.g. numbers like 2,3 and defined constants in a code

- **Expression**: match patterns on constants and complex subterms
  e.g. \(\text{struct} \rightarrow \text{elem}, \ x - y, \ \text{func}(\text{arg}) \) etc

- **Identifier**: a structure field, a macro, a function, or a variable
Metavariables

Example: $x \rightarrow y = m \rightarrow n + 1$;

- **Constant**: match patterns on values and constants
  
  e.g. numbers like 2, 3 and defined constants in a code

- **Expression**: match patterns on constants and complex subterms
  
  e.g. `struct -> elem, x-y, func(arg)` etc.

- **Identifier**: a structure field, a macro, a function, or a variable

- **Statement**: match patterns which do not return a value
  
  e.g. `if, while, break` etc
Metavariables

- **Constant**: match patterns on values and constants
e.g. numbers like 2, 3 and defined constants in a code

- **Expression**: match patterns on constants and complex subterms
e.g. `struct->elem`, `x-y`, `func(arg)`

- **Identifier**: a structure field, a macro, a function, or a variable

- **Statement**: match patterns which do not return a value
e.g. `if`, `while`, `break` etc

- **Type**: match patterns for the type of variables/functions
e.g. `int`, `boolean`, `float` etc
Transformation specification

- in the leftmost column for something to remove

+ in the leftmost column for something to add

* in the leftmost column for something of interest
  - Cannot be used with + and -.

Spaces, newlines that are irrelevant.
Spatch

- Coccinelle’s command-line tool

- To check that your semantic patch is valid:
  
  spatch --parse-cocci mysp.cocci

- To run your semantic patch:
  
  spatch --sp-file mysp.cocci file.c

  spatch --sp-file mysp.cocci --dir directory
Exercise 1

- Save the semantic patch to bitmask.cocci. [slide 11 and 13]

- Run it using spatch on any particular directory or on whole kernel.
  `spatch --sp-file bitmask.cocci --dir directory`

- Redirect results to an output file for an inspection.

- Is it ok to use BIT macro in every case? Should we want to restrict it for the files which are already using it?
Exercise 2

- Parentheses are not needed around the bitwise left shift operations like in `u32 val = (1 << 31);`.

- Write a semantic patch to remove these parentheses.

- Run the semantic patch over the directory `drivers/net/wireless/`.

- Some other cases to think about:
  - Extra parentheses around the function arguments
  - Using the same identifier on the left and right side of the assignment
Using BIT macro (Revisited)

- **Example:**

```diff
diff -u -p a/arch/mips/pci/pci-mt7620.c b/arch/mips/pci/pci-mt7620.c
--- a/arch/mips/pci/pci-mt7620.c
+++ b/arch/mips/pci/pci-mt7620.c
@@ -37,11 +37,11 @@
    #define PDRV_SW_SET BIT(23)

    #define PPLL_DRV 0xa0
    -#define PDRV_SW_SET (1<<31)
    -#define LC_CKDRVPD (1<<19)
    -#define LC_CKDRVOHZ (1<<18)
    -#define LC_CKDRVHZ (1<<17)
    -#define LC_CKTEST (1<<16)
    +#define PDRV_SW_SET (BIT(31))
    +#define LC_CKDRVPD (BIT(19))
    +#define LC_CKDRVOHZ (BIT(18))
    +#define LC_CKDRVHZ (BIT(17))
    +#define LC_CKTEST (BIT(16))
```
Using BIT macro (Revisited)

**Example:**

```diff
--- a/arch/mips/pci/pci-mt7620.c
+++ b/arch/mips/pci/pci-mt7620.c
@@ -37,11 +37,11 @@

 #define PDRV_SW_SET BIT(23)

 #define PPLL_DRV 0xa0
-#define PDRV_SW_SET (1<<31)
-#define LC_CKDRVPD (1<<19)
-#define LC_CKDRVVOHZ (1<<18)
-#define LC_CKDRVHZ (1<<17)
-#define LC_CKTEST (1<<16)
+#define PDRV_SW_SET (BIT(31))
+#define LC_CKDRVPD (BIT(19))
+#define LC_CKDRVVOHZ (BIT(18))
+#define LC_CKDRVHZ (BIT(17))
+#define LC_CKTEST (BIT(16))
```

- Would like to restrict the bitmask semantic patch to files that are already using the BIT macro?
Using BIT macro (Revisited)

Example:

- #define LC_CKDRVPD (1<<19)
- #define LC_CKDRVOHZ (1<<18)
+ #define LC_CKDRVPD (BIT(19))
+ #define LC_CKDRVOHZ (BIT(18))

Semantic patch:

@usesbit@
@@
BIT(...)
@depends on usesbit@
expression E;
@@
- 1 << E
+ BIT(E)
Isomorphism

- Coccinelle captures code as defined in your rule
- Valid variants of your defined pattern can exist
- Cumbersome to list them all in your rule/s

Examples:
- \( x == \text{NULL} \) and \( !x \)
- \( \text{sizeof(struct i)} * e \) and \( e * \text{sizeof(struct i)} \)

- Isomorphisms can handle such variations
- Rules defining isomorphisms exist in standard.iso
Isomorphism Examples

Example 1:

Expression

@ is_null @
expression X;
@@

X == NULL <=> NULL == X => !X

Example 2:

Expression

@ drop_cast @
expression E;
pure type T;
@@

(T)E => E
Exercise 3

- Consider the example of DIV_ROUND_UP.

- The macro is defined in linux/kernel.h. So, it depends on this header file.

- Expand the semantic patch you wrote in exercise 2 using 'depends on'.

- Review the output given by updated semantic patch.
Exercise 4

- To avoid code duplication or error prone code, the kernel provides macros such as DIV_ROUND_UP.

- The definition of the DIV_ROUND_UP goes like this:
  \[
  \text{DIV_ROUND_UP}(n, d) = \left( \left( n + d - 1 \right) / d \right)
  \]

- Write the semantic patch for replacing the pattern \((\left( n + d - 1 \right) / d)\) with \text{DIV_ROUND_UP}.

- Redirect results to an output file for an inspection.
Example: setup_timer

- The function setup_timer combines the initialization of a timer with the initialization of the timer's function and data fields.

```
- init_timer(&cf->timer);
- cf->timer.function = omap_cf_timer;
- cf->timer.data = (unsigned long) cf;
+ setup_timer(&cf->timer, omap_cf_timer, (unsigned long)cf);
```

- Why setup_timer?

- How Coccinelle can help here?
setup_timer: case one

Example:

@@
@@
- init_timer(&cf->timer);
- cf->timer.function = omap_cf_timer;
- cf->timer.data = (unsigned long) cf;
+ setup_timer(&cf->timer, omap_cf_timer, (unsigned long)cf);

Semantic patch

@case_one@
expression e, func, da;
@@

- init_timer (&e);
+ setup_timer (&e, func, da);
- e.function = func;
- e.data = da;
setup_timer: case one

Semantic patch:

@case_one@
expression e, func, da;
@@

- init_timer (&e);
+ setup_timer (&e, func, da);
- e.function = func;
- e.data = da;

- Is this the only case where we can use setup_timer?

- Is it necessary that the call to init_and the initialization of the function and data fields always occur in the order shown in the example?
setup_timer: case two

Example:

- init_timer(&hose->err_timer);
- hose->err_timer.data = (unsigned long)hose;
- hose->err_timer.function = pcibios_enable_err;
+ setup_timer(&hose->err_timer, pcibios_enable_err, (unsigned long)hose);

Semantic patch:

@case_two@
expression e, func, da;
@@
- init_timer (&e);
+ setup_timer (&e, func, da);
- e.data = da;
- e.function = func;
setup_timer: comparing both cases

Case one:

```c
@case_one@
expression e, func, da;
@@
- init_timer (&e);
+ setup_timer (&e, func, da);
- e.function = func;
- e.data = da;
```

Case two:

```c
@case_two@
expression e, func, da;
@@
- init_timer (&e);
+ setup_timer (&e, func, da);
- e.data = da;
- e.function = func;
```
Disjunctions

- A sequence of patterns between ( ... | ... ).

- Patterns checked in order and the first that matches is chosen.

- Combining case one and case two in our example:

```c
@case_one_and_two@
expression e, func, da;
@@
-init_timer (&e);
+setup_timer (&e, func, da);
(
- e.function = func;
- e.data = da;
|  
- e.data = da;
- e.function = func;
)```
Exercise 5

- Implement the semantic patches for both cases of the setup_timer. Compare the results.

- Implement the rule combining case one and case two using disjunction.

- Think about why do we need to use disjunctions? Can we use multiple rules?

- Check the results. Does it cover all the cases that were matched by the separate rules?

- Grep for the init_timer and check if the rule with disjunction covers everything?
setup_timer(Contd.)

Example:

```c
init_timer (&np->timer);
np->timer.expires = jiffies + 1*HZ;
np->timer.data = (unsigned long) dev;
np->timer.function = rio_timer;
add_timer (&np->timer);
```

- Does previous rule covered all cases?

- Is it necessary that the call to init_timer and the initialization of the function & the data field always occurs in a contiguous manner?
**Dots**

**Problem:**

- Sometimes it is necessary to search for multiple related code fragments.

**Solution:**

- Specify patterns consisting of the fragments of code separated by arbitrary execution paths.
- Specify constraints on the contents of those execution paths.
setup_timer: case three

Semantic patch:

```c
@case_three@
expression e, func, da;
@@

- init_timer (&e);
+ setup_timer (&e, func, da);
    ...

- e.data = da;
- e.function = func;
```

Example:

```c
- init_timer (&np->timer);
+ setup_timer(&np->timer, rio_timer, (unsigned long)dev);
    np->timer.expires = jiffies + 1*HZ;
- np->timer.data = (unsigned long) dev;
- np->timer.function = rio_timer;
    add_timer (&np->timer);
```
Using dots

Semantic patch:

```c
@case_three@
expression e, func, da;
@@
- init_timer (&e);
+ setup_timer (&e, func, da);
...  
- e.data = da;
- e.function = func;
```

- `'...' matches all possible execution paths from the pattern before to the pattern after

- The patterns before and after cannot appear in the region matched by “...” (shortest path principle).
Example: Compressing lines for immediate return

- In the following code last two lines could be compressed into one:

```c
int bytes_written;
uint16_t link_speed;

link_speed = rtw_get_cur_max_rate(padapter) / 10;
bytes_written = snprintf(command, total_len, "LinkSpeed %d", link_speed);
return bytes_written;
```
Compressing lines for immediate return

- In the following code last two lines could be compressed into one:

```c
int bytes_written;
ul6 link_speed;

link_speed = rtw_get_cur_max_rate(padapter) / 10;
bytes_written = snprintf(command, total_len, "LinkSpeed %d", link_speed);
return bytes_written;
```
Dots: Compressing lines for immediate return

Example:

```c
- bytes_written = snprintf(command, total_len, "LinkSpeed %d",
+ return snprintf(command, total_len, "LinkSpeed %d",
                  link_speed);
- return bytes_written;
```

Semantic patch:

```c
@@
extension r;
identifier f;
@@
-r = f(...)
+return
  f(...);
-return r;
```
Exercise 6

- Implement the rule for case three of setup_timer using dots. [Slide 40]

- Run the patch over the kernel code and investigate the result.

- Think about the case three like pattern for the case two.

- Implement the rule for those kind of patterns.

- Try to limit the number of rules.
Exercise 6 (Contd.)

Example:

```c
init_timer(&sharpsl_pm.ac_timer);
sharpsl_pm.ac_timer.function = sharpsl_ac_timer;

init_timer(&sharpsl_pm.chrg_full_timer);
sharpsl_pm.chrg_full_timer.function = sharpsl_chrg_full_timer;
```

- Is it even necessary that the initialization of the data field always occurs?
- Expand the semantic patch to include such cases.
Exercise 7

Example:

```c
int bytes_written;
uint16_t link_speed;

link_speed = rtw_get_cur_max_rate(padapter) / 10;
return snprintf(command, total_len, "LinkSpeed %d", link_speed);
```

- Do we really need the variable `bytes_written` after compressing the lines?

- Expand the semantic patch[slide 44] to remove the variable along with compressing lines.

  Hint: Ensure that the variable is not used anywhere else.
Using dots (Contd.)

Semantic patch:

```c
@case_three@
expressions e, func, da;
@@
- init_timer (&e);
+ setup_timer (&e, func, da);
  ...
- e.data = da;
- e.function = func;
```

- Check the properties of the matched statement sequence
- Does the rule look correct? Or do we need to ensure something?
Using dots with when

- Dots can be modified with a when clause, indicating a pattern that should not occur

```c
@case_three@
expression e1, e2, e3, e4, func, da;
@@
-init_timer(&e1);
+setup_timer(&e1, func, da);
... when ! func = e2
  when ! da = e3
-e1.data = da;
-e1.function = func;
```
when

- Keyword used to indicate conditions on execution path
- As seen before, controls the behavior of “...”
- Can be coupled with:
  - strict: force condition on every execution path (including failures)
  - forall: force condition on every execution path (excluding failures)
  - exists: is there an execution path that matches the pattern?
  - any: allow the patterns specified...
  - conditions specified by the user
More use of dots

- Two possible modifiers to the control flow for ellipses:

  1. `<...P...>` indicates that matching the pattern in between the ellipses is optional

  2. `<+...P...+>` indicates that the pattern in between the ellipses must be matched at least once, on some control-flow path.

    - The + is intended to be reminiscent of the + used in regular expressions.
More use of dots (Contd.)

Example:

```c
@if (...) {
 <+...
     return ...;
  ...+>
}
```

Meaning:

- To remove all ifs that contain at least one return.
More use of dots (Contd.)

Example:

```c
if (...) {
    return ...;
    ...
}
```

Meaning:

- To remove all ifs
Exercise 8

1. Implement the example of 'compression of lines for the immediate return problem'.

2. The semantic patch for removing unused variables only matches a variable declaration when the declaration does not initialize the variable.

3. Extend the complete semantic patch so that it also removes unused variables that are initialized to a constant.
Exercise 9

In the following code, when \( x \) has any pointer type, the cast to \( u8 \ast \), or to any other pointer type is not needed.

\[
kfree((u8 \ast)x);
\]

- Write a semantic patch to remove such casts.

- Consider generalizing your semantic patch to functions other than \( kfree \).

- Are there any patterns that can benefit from using disjunctions?
Coccicheck

- A Coccinelle-specific target which is defined in the top level Makefile.

- Four basic modes
  - Patch mode
  - Context mode
  - Org mode
  - Report mode

- Default output: Report mode

- Command that can be used for specifying particular mode:
  `make coccicheck MODE=patch`
Modes for the Coccinelle script

- Four basic modes
  - Patch mode: proposes a fix when possible.

@@ -582,8 +580,7 @@ static int iss_net_configure(int index,
   return 1;
 }

- init_timer(&lp->tl);
- lp->tl.function = iss_net_user_timer_expire;
+ setup_timer(&lp->tl, iss_net_user_timer_expire, 0UL);

   return 0;
Modes for the Coccinelle script

- **Four basic modes**

- **Context mode:**
  1. highlights lines of interest and their context in a diff-like style.
  2. Lines of interest are indicated with '-'.

```c
@@ -582,8 +580,7 @@ static int iss_net_configure(int index,
    return 1;
 }
-    init_timer(&lp->tl);
-    lp->tl.function = iss_net_user_timer_expire;
-    setup_timer(&lp->tl, iss_net_user_timer_expire, 0UL);

    return 0;
```
Modes for the Coccinelle script

• **Four basic modes**

  - Org mode: Generates a report in the Org mode format of Emacs.

```plaintext
* TODO [[view:/home/linux-next/linux/arch/sh/drivers/pci/common.c::face=ovl-face1::cole=12] [Use setup_timer function.]]
  [[view:/home/linux-next/linux/arch/sh/drivers/pci/common.c::face=ovl-face1::linb=[/home/linux-next/linux/arch/sh/drivers/pci/common.c::109]]

* TODO [[view:/home/linux-next/linux/arch/sh/drivers/pci/common.c::face=ovl-face1::cole=12] [Use setup_timer function.]]
  [[view:/home/linux-next/linux/arch/sh/drivers/pci/common.c::face=ovl-face1::linb=[/home/linux-next/linux/arch/sh/drivers/pci/common.c::115]]
```
Modes for the Coccinelle script

- Four basic modes
  - Report mode: Generates a list in the following format
    file:line:column-column: message

    /home/linux-next/linux/arch/sh/drivers/pci/common.c:108:2-12: Use setup_timer function
    /home/linux-next/linux/arch/sh/drivers/pci/common.c:114:2-12: Use setup_timer function
    /home/linux-next/linux/arch/x86/kernel/pci-calgary_64.c:1010:1-11: Use setup_timer function around line 1011.
setup_timer again

**Problem:**

- What if `init_timer` is called in one function and data field is initialized in another function?

- Will it be safe to use `setup_timer` in that case?

**Solution:**

- How about giving warning in such cases?
setup_timer again

- We need two rules to match both parts

Semantic patch:

```c
@r1@
identifier f;
@@

f(...) { ...
    init_timer(...)
    ...}
@r2@
identifier g;
struct timer_list t;
expression e;
@@

g(...) { ...
    t.data = e
    ...}
```
setup_timer again

- We want to match 2 different functions. So, let's avoid function name overriding.

Semantic patch:

```c
@r1 exists identifier f;
@@

f(...) { ...
    init_timer(...)
    ...
}@@

@r2 exists identifier g != r1.f;
struct timer_list t;
expression e;
@@

g(...) { ...
    t.data = e
    ....
}@@
```
Position variables

- Position metavariables can be used to store the position of any token, for later matching or printing.

- In the case of `setup_timer` we want to use the position of `init_timer` so that Coccinelle can give warning at such code.
Position variables

Example:

```c
@r1 exists@
identifier f;
position p;
@@

f(...) { ...
    init_timer@p(...)
    ...
}

@r2 exists@
identifier g != r1.f;
struct timer_list t;
expression e8;
@@

g(...) { ...
    t.data = e8
    ...
}
```
Embedding python script

- 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.
Python script with the warning

Example:

```python
@r1 exists@
identifier f;
position p;
@@
f(...) { ...
   init_timer@p(...)  
   ...
}

@r2 exists@
identifier g != r1.f;
struct timer_list t;
expression e;
@@
g(...) { ...
   t.data = e
   ...
}
@script:python depends on r2@
p << r1.p;
@@
print "Data field initialized in another function. Dangerous to use
call setup_timer %s:%s" % (p[0].file,p[0].line)```
**Python script without printing warning**

**Example:**

```python
@r1 exists
identifier f;
position p;
@@
f(...) { ...
    init_timer@p(...)
    ...
}

@r2 exists
identifier g != r1.f;
struct timer_list t;
expression e;
@@
g(...) { ...
    t.data = e
    ...
}

@script:python  depends on  r2@
p << r1.p;
@@
cocci.include_match(False)
```
Exercise 10

- When searching for things, rather than transforming them, it may be useful to generate the output in a variety of formats. This can be done using the interface to python (ocaml is also available).

- Position variables are useful in this context, because they provide the file name and line number of various program elements.
Exercise 10 (Contd.)

- Consider the following patch discussed earlier:

```python
@@
expression r;
identifier f;
@@
-r = f(...)
+return
    f(...);
-return r;
```

- Following python code is intended to print the file name and line numbers of the assignment and erroneous test, respectively:

```python
@script:python@
p1 << r.p1; // inherit a metavariable p1 from rule r
p2 << r.p2; // inherit a metavariable p2 from rule r
@@
print p1[0].file, p1[0].line, p2[0].line
```
Exercise 10 (Contd.)

Do this:

- Create a semantic patch consisting of the original patch rule shown on the previous page followed by the python code specified in the last slide.

- Give name r to the rule and remove the transformation.

- Add position variables p1 and p2.

- Attach position variables to the relevant code.

- Test the semantic patch and investigate the results.
Exercise 11

- We have seen that * can be used to highlight items of interest.

- Repeat the previous exercise, this time without using python, but instead annotate the original code pattern with * rather than performing transformations.

- How is the result different than the result produced when using python?
Exercise 12

• Implement the setup_timer case with the python code.

• Combine all rules in a single script and then try to run it. Observe how output changes.

• Try to reorder the rules in a semantic patch and then observe the changes.

• Do we also need a rule for the immediate call of init_timer, initialization of data and function fields? If yes, then why? If no, then why?

Hint: Consider performance and speed of the semantic patch.
Feature summary

• Metavariables and Isomorphisms

• Different uses of ...

• When

• Named rules and metavariable inheritance

• Position variables

• Scripting through Python/Ocaml

• Different modes for the Coccinelle script
Useful links

- **Source code of the Coccinelle**: "https://github.com/coccinelle/coccinelle"
- **Grammar and features**: "http://coccinelle.lip6.fr/docs/options.pdf"
- **Documentation**: "Documentation/coccinelle.txt"
- **Project**: "http://coccinelle.lip6.fr/"
- **Spgen**: "https://github.com/coccinelle/coccinelle/tree/master/tools/spgen"

THANK YOU!
Acknowledgement

- **Julia Lawall** [Developer and maintainer of Coccinelle]
- **Aya Mahfouz** [Outreachy intern, round 9]