Coccinelle Features

Julia Lawall (Inria/LIP6)

http://coccinelle.lip6.fr

June 25, 2018

Coccinelle features

- Isomorphisms.
- Depends on.
- Positions.
- Python.
- Nests.

Isomorphisms

Issue:

- Coccinelle matches code exactly as it appears.
- `x == NULL` does not match `!x`.

Goal:

- Transparently treat similar code patterns in a similar way.

Example: DIV_ROUND_UP

The following code is fairly hard to understand:

```
return (time_ns * 1000 + tick_ps - 1) / tick_ps;
```

kernel.h provides the following macro:

```
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))
```

This is used, but not everywhere it could be.

We can write a semantic patch to introduce new uses.

DIV_ROUND_UP semantic patch

One option:

```
@@ expression n,d; @@

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

Another option:

```
@@ expression n,d; @@

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

Problem: How many parentheses are needed, to capture all occurrences?

Isomorphisms

An isomorphism relates code patterns that are considered to be similar:

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

 (T)E => E

Expression
@ paren @ expression E; @@

 (E) => E

Expression
@ is_null @ expression X; @@

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

Isomorphisms, contd.

Isomorphisms are handled by rewriting.

```
(((n) + (d) - 1) / (d))
```

becomes:

```
(
 (((n) + (d) - 1) / (d))
|
 (((n) + (d) - 1) / d)
|
 (((n) + d - 1) / (d))
|
 (((n) + d - 1) / d)
|
 ((n + (d) - 1) / (d))
|
 ((n + (d) - 1) / d)
|
 ((n + d - 1) / (d))
|
 ((n + d - 1) / d)
|
 etc.
)
```

Results

```
@@
expression n,d;
@@

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

Changes 281 occurrences in Linux 3.2.

Practical issues

Default isomorphisms are defined in standard.iso

To use a different set of default isomorphisms:

```
spatch --sp-file mysp.cocci --dir linux-x.y.z --iso-file empty.iso
```

To drop specific isomorpshisms:

```
@disable paren@ expression n,d; @@
- (((n) + (d) - 1) / (d))
+ DIV_ROUND_UP(n,d)
```

To add rule-specific isomorphisms:

```
@using "myparen.iso" disable paren@
expression n,d;
@@
- (((n) + (d) - 1) / (d))
+ DIV_ROUND_UP(n,d)
```

Exercise 6

Some Linux code combines an assignment with a test, as illustrated by the following:

```
if (!(p = kmalloc(sz, GFP_KERNEL)))
  break;
```

The following semantic patch moves the assignment out of the conditional:

```
@@ identifier e1; expression e2; statement S1, S2; @@
+ e1 = e2;
  if (
-     (e1 = e2)
+     e1
      == NULL) S1 else S2
```

1. Test this semantic patch on linux-3.2/sound/pci/au88x0
2. How were isomorphisms used in these matches?

Exercise 7

Run

```
spatch --parse-cocci sp.cocci
```

For some semantic patch `sp.cocci` that you have developed.

Explain the result.

Depends on

```
@@
expression n,d;
@@

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

Issue:

- `DIV_ROUND_UP` is a macro, defined in kernel.h.
- Maybe some file does not include kernel.h?
- `#include`, if present, would not be in the same function.

Depends on, contd.

```
@r@
@@
#include <linux/kernel.h>

@depends on r@
expression n,d;
@@
- (((n) + (d) - 1) / (d))
+ DIV_ROUND_UP(n,d)
```

Results:

- Naming a rule lets it be referenced by other rules.
- Only introduce `DIV_ROUND_UP` if the `#include` rule is satisfied.
- Matches 86 occurrences.

Positions and Python

```
@@ expression e,e1; identifier f; @@
  e = kmalloc(...);
  ... when != e = e1
(
  e == NULL || ...
|
  e != NULL && ...
|
* e->f
)
```

Output reported as a diff:

- Useful in emacs (diff-mode).
- Perhaps less useful in other contexts.

Bonus question: Why is there no * on `kmalloc`?

Positions and Python

Goal:

- Collect positions of some matched elements.
- Print a helpful error message.

```
@r@
expression e,e1;
identifier f;
position p1, p2;
@@
  e = kmalloc@p1(...);
  ... when != e = e1
(
  e == NULL || ...
|
  e != NULL && ...
|
  e@p2->f
)
```

```
@script:python@
p1 << r.p1;
p2 << r.p2;
@@
l1 = p1[0].line
l2 = p2[0].line
print "kmalloc on line %s not tested
       before reference on line %s" %
       (l1,l2)
```

A refinement

Exists:

- Require only a single matching execution path.
- Default for *.

```
@r exists@
expression e,e1;
identifier f;
position p1, p2;
@@
  e = kmalloc@p1(...);
  ... when != e = e1
(
  e == NULL || ...
|
  e != NULL && ...
|
  e@p2->f
)
```

```
@script:python@
p1 << r.p1;
p2 << r.p2;
@@
l1 = p1[0].line
l2 = p2[0].line
print "kmalloc on line %s not tested
       before reference on line %s" %
       (l1,l2)
```

Exercise 8

Rewrite a semantic patch that you have implemented previously, so that it prints the line numbers on which a change is needed, rather than making the change.

Useful terms:

- `p[0].file` is the name of the file represented by `p`.
- `p[0].line` is the number, as a string, of the line represented by `p`.
- `p` is an array, because there can be many matches.

Nests

A problem:

Linux provides generic error functions and specific ones:

- `pr_debug("%c is not valid", c);`:
  - Generic
- `netdev_dbg(dev->net, "registers:");;`:
  - Netdev specific

When to use `netdev_dbg`?

Choosing netdev_dbg

netdev_dbg requires a struct net_device * argument.

- Such an argument may be accessible from a parameter of the enclosing function.

```
static void nc_dump_registers(struct usbnet *dev) {
        u8      reg;
        u16     *vp = kmalloc(sizeof (u16));

        if (!vp)
                return;

        netdev_dbg(dev->net, "registers:");
        ...
}
```

Still, the net field is not apparent...

Semantic patch

Strategy:

- Find a function that contains a call.
- Find the type definition of each of its parameters.
- If a parameter has a type with a `net_device` field, transform the call.

Step 1:

```
@r exists@ identifier f,s,i; @@

f(...,struct s *i,...) {
  <+...
  pr_debug(...)
  ...+>
}
```

Nest `<+... P ...+>` checks whether P occurs at least once.

Solution, contd.

```
@rr@
identifier r.s,fld;
@@

struct s {
  ...
  struct net_device *fld;
  ...
};

@@
identifier r.f,r.s,r.i,rr.fld;
@@

f(...,struct s *i,...) {
  <...
- pr_debug
+ netdev_dbg
  (
+ i->fld,
  ...)
  ...>
}
```

Optimization

Observation: Too bad to have to match `f` all over again.

- Remember the positions of the calls that were matched.
- Only update the calls in those positions.

Step 1:

```
@r exists@
identifier f,s,i;
position p;
@@

f(...,struct s *i,...) {
  <+...
  pr_debug@p(...)
  ...+>
}
```

Optimization, contd.

```
@rr@
identifier r.s,fld;
@@

struct s {
  ...
  struct net_device *fld;
  ...
};

@@
identifier r.i;
position r.p;
identifier rr.fld;
@@

- pr_debug@p
+ netdev_dbg
    (
+   i->fld,
    ...)
```

Exercise 9

1. The following semantic patch rule matches an initialization of a platform driver structure:

```
@platform@
identifier p, probefn, removefn;
@@
struct platform_driver p = {
  .probe = probefn,
  .remove = removefn,
};
```

   Extend this semantic patch to find the name of the first parameter of the probe function and of the remove function, and to print the function names and the corresponding parameter names using python.

2. Some platform driver probe functions use the function `kzalloc` to allocate memory. Adjust the previous semantic patch to find and print the positions of these calls.

Exercise 9, contd.

3. `Kzallocd` memory must be freed using `kfree`, but this is easy to forget. The function `devm_kzalloc` is like `kzalloc` but the driver library manages the freeing. Adjust the previous semantic patch to replace calls to `kzalloc` by calls to `devm_kzalloc`.

4. The code produced by the previous rule does not compile, because `devm_kzalloc` requires a device argument. This can be constructed from the first parameter of a platform driver probe function. If this parameter is named `x`, then the corresponding device value is `&x->dev`. Adjust the previous semantic patch to add this as the first argument of each generated call to `devm_kzalloc`.

Exercise 9, contd.

5. The code resulting from the previous semantic patch has double frees, because `devm_kzalloc` causes an implicit free, and the code still contains calls to `kfree`. How can you solve this problem?

Summary

- Isomorphisms, for simplifying, eg NULL tests, parentheses, casts.
- Positions, for remembering the exact position of some code.
- Python, for printing error messages, managing hashtables, etc.
- Nests, for matching any number of something.