Introduction to Coccinelle

Julia Lawall
University of Copenhagen

November 11, 2011

Overview

- The structure of a semantic patch.
- Dots.
- Nests.
- Isomorphisms.
- Depends on.
- Positions.
- Python.

The structure of a semantic patch

Goals:

- Specify patterns of code to be found and transformed.
- Specify which terms should be abstracted over.
- C-like, patch-like notation.

```
diff -u -p a/drivers/usb/usb-skeleton.c b/drivers/usb/us
--- a/drivers/usb/usb-skeleton.c 2011-10-19 23:44:50.344
+++ b/drivers/usb/usb-skeleton.c 2011-11-10 19:57:05.148
@@ -358,7 +358,7 @@ retry:
                rv = skel_do_read_io(dev, count);
                if (rv < 0)
                        goto exit;
-               else if (!file->f_flags & O_NONBLOCK)
+               else if (!(file->f_flags & O_NONBLOCK))
                        goto retry;
                rv = -EAGAIN;
        }
```

The !& problem

The problem: Combining a boolean (0/1) with a constant using & is usually meaningless:

```
if(!erq->flags & IW_ENCODE_MODE)
{
        return -EINVAL;
}
```

The solution: Add parentheses.

Our goal: Do this automatically for any expression E and constant C.

A semantic patch for the !& problem

```
@@
expression E;
constant C;
@@

  !E & C
```

Two parts per rule:

- Metavariable declaration
- Transformation specification

A semantic patch can contain multiple rules

A semantic patch for the !& problem

```
@@
expression E;
constant C;
@@

- !E & C
+ !(E & C)
```

Two parts per rule:

- Metavariable declaration
- Transformation specification

A semantic patch can contain multiple rules

Exercise 1

1. Create a file ex1.cocci containing the following:

```
@@
expression E;
constant C;
@@

- !E & C
+ !(E & C)
```

2. Create a file ex1.c that contains some valid and invalid uses of `!` and `&`
3. Run spatch: `spatch -sp_file ex1.cocci ex1.c`
4. Did your semantic patch do everything it should have?
5. Did it do something it should not have?

Practical issues

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
```

To understand why your semantic patch didn't work:

```
spatch -sp_file mysp.cocci file.c -debug
```

Metavariable types

- expression, statement, type, constant, local idexpression
- A type from the source program
- iterator, declarer, iterator name, declarer name, typedef

Example:

```
@@
unsigned int E;
constant C;
@@

- !E & C
+ !(E & C)
```

Transformation specification

- − in the leftmost column for something to remove
- + in the leftmost column for something to add
- $\star$ in the leftmost column for something of interest
  - Cannot be used with + and −.
- Spaces, newlines irrelevant.

```
@@ expression E; constant C; @@
! + ( E & C + )
```

The `sizeof` problem

In C, `sizeof` can take two kinds of argument:

- A type: `sizeof(char)` = 1
- An expression: Suppose `c` has type `char *`.
  - `sizeof(c)` = 4
  - `sizeof(*c)` = 1

A common problem is to take the size of a pointer, rather than the size of the referenced structure:

```
  memset(blkbk->pending_reqs, 0,
-          sizeof(blkbk->pending_reqs));
+          sizeof(*blkbk->pending_reqs));
```

Exercise 2

1. Write a semantic patch, ex2.cocci, to find and fix incorrect uses of `sizeof`.
   - Hint: A metavariable declared as `expression *e;` can only match a pointer-typed expression.
2. Write a test file, ex2.c, containing various uses of sizeof.
3. Use `spatch -sp_file ex2.cocci ex2.c` to test your semantic patch on your code.
4. Write another semantic patch, ex2a.cocci, that uses * to find occurrences of the problem, but not change the code.
5. Test ex2a.cocci on ex2.c

Exercise 3

Write rules to introduce calls to the following functions:

```
static inline void *
ide_get_hwifdata (ide_hwif_t * hwif)
{
        return hwif->hwif_data;
}

static inline void
ide_set_hwifdata (ide_hwif_t * hwif, void *data)
{
        hwif->hwif_data = data;
}
```

Hints:

- To only consider `ide_hwif_t`-typed expressions, declare a "metavariable" `typedef ide_hwif_t;`.
- Consider both structures and pointers to structures.
- Consider the ordering of the rules.

Solution 1

```
@@
typedef ide_hwif_t;
ide_hwif_t *dev;
expression data;
@@

- dev->hwif_data = data
+ ide_set_hwifdata(dev,data)


@@
ide_hwif_t *dev;
@@

- dev->hwif_data
+ ide_get_hwifdata(dev)
```

Solution 2 (more concise)

```
@@
ide_hwif_t *dev;
expression data;
@@

(
- dev->hwif_data = data
+ ide_set_hwifdata(dev,data)
|
- dev->hwif_data
+ ide_get_hwifdata(dev)
)
```

Solution 3 (more complete)

```
@@ ide_hwif_t *dev; expression data; @@
(
- dev->hwif_data = data
+ ide_set_hwifdata(dev,data)
|
- dev->hwif_data
+ ide_get_hwifdata(dev)
)

@@ ide_hwif_t dev; expression data; @@
(
- dev.hwif_data = data
+ ide_set_hwifdata(&dev,data)
|
- dev.hwif_data
+ ide_get_hwifdata(&dev)
)
```

Dots

Goals:

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

Nested spin_lock_irqsave

`spin_lock_irqsave(lock,flags)`:

- Takes a lock.
- Saves current interrupt status in `flags`.
- Disables interrupts.

Invalid nested usage:

```
spin_lock_irqsave(&port->lock, flags);
if (sx_crtscts(port->port.tty))
  if (set & TIOCM_RTS) port->MSVR |= MSVR_DTR;
 else if (set & TIOCM_DTR) port->MSVR |= MSVR_DTR;
spin_lock_irqsave(&bp->lock, flags);
sx_out(bp, CD186x_CAR, port_No(port));
sx_out(bp, CD186x_MSVR, port->MSVR);
spin_unlock_irqrestore(&bp->lock, flags);
spin_unlock_irqrestore(&port->lock, flags);
```

Detecting nested spin_lock_irqsave

Observations:

- Calls to `spin_lock_irqsave` share their second argument.
  - Solution: repeated metavariables.
- Calls to `spin_lock_irqsave` may be separated by arbitrary code.
  - Solution: ...
- There should be no calls to `spin_lock_irqrestore` between the calls to `spin_lock_irqsave`.
  - Solution: when

A semantic match for detecting nested spin_lock_irqsave

```
@@
expression lock1,lock2;
expression flags;
@@

*spin_lock_irqsave(lock1,flags)
... when != flags
*spin_lock_irqsave(lock2,flags)
```

Exercise: NULL pointer dereferences

The Linux kernel function `kmalloc` returns `NULL` if the allocation fails.

- The result of `kmalloc` should not be dereferenced without first checking for `NULL`.

Example:

```
g = kmalloc (sizeof (*g), GFP_KERNEL);
g->next = chains[r_sym].next;
```

Exercise: Write a semantic match that detects this problem.

Another source of NULL pointer dereferences

```
if (!wl) {
  wiphy_err(wl->wiphy,
            "brcms_suspend: pci_get_drvdata failed");
  return -ENODEV;
}
```

Observation: `wl` is NULL inside the “then” branch

- It may be useful to be informed of all of the dereferences.

Nests

Goals:

- Describe terms that can occur any number of times within an execution path.
- 0 or more times:

  $$<\ldots \quad P \quad \ldots>$$

- 1 or more times:

  $$<+\ldots \quad P \quad \ldots+>$$

Exercise: Write a semantic patch to detect dereferences under a `NULL` test.

Isomorphisms

Goals:

- Transparently treat similar code patterns in a similar way.

Examples:

```
if (!wl) { ... }

if (wl == NULL) { ... }
```

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 to put, to capture all occurrences?

Isomorphisms

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

```
Expression
@ is_null @ expression X; @@

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

Expression
@ paren @ expression E; @@

  (E) => E

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

  (T)E => E
```

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.
)
```

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

Run

```
spatch -parse_cocci sp.cocci
```

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

Explain the result.

Depends on

Goals:

- Define multiple matching and transformation rules.
- Express that the applicability of one rule depends on the success or failure of another.

Header files

DIV_ROUND_UP is defined in kernel.h

- The transformation might not be correct if kernel.h is not included.
- Problem: #include <linux/kernel.h> is far from the call to DIV_ROUND_UP

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

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

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

Positions and Python

Goals:

- Positions: remember exactly what fragment of code was matched.
- Python: do arbitrary computation, especially printing.

& with 0

```
if (mode & V4L2_TUNER_MODE_MONO)
  s1 |= TDA8425_S1_STEREO_MONO;
```

- `V4L2_TUNER_MODE_MONO` is 0.
- The test is always false.

Detecting & with 0

One strategy:

- Find a use of &.
- Check that the constant is 0.
- Check that there is not another nonzero definition.
- Report on the bug site.

Find a use of &

```
@r expression@
identifier C;
expression E;
position p;
@@

 E & C@p
```

- The rule has a name: `r`.
- `p` is a position metavariable, so we can find the same & expression later.

Check that `C` is 0

```
@s@
identifier r.C;
@@
#define C 0

@t@
identifier r.C;
expression E != 0;
@@
#define C E
```

- Both rules inherit `C`.
- Each rule is applied once for each value of `C`.
- The second rule puts a constraint on `E`.
  - Constraints on constants, expressions, identifiers, positions
  - Regular expressions allowed for constants and identifiers.

Printing the result

```
@script:python depends on s && !t@
p << r.p;
C << r.C;
@@

cocci.print_main("and with 0", p)
```

- Python rules only inherit metavariables, using << notation.
- Depends on clause is evaluated for each inherited set of metavariable bindings.
- print_main is part of a library for printing output in Emacs org mode.

The complete semantic patch

```
@r expression@
identifier C;
expression E;
position p;
@@
E & C@p

@s@ identifier r.C; @@
#define C 0

@t@ identifier r.C; expression E != 0; @@
#define C E

@script:python depends on s && !t@
p << r.p;
C << r.C;
@@
cocci.print_main("and with 0", p)
```

Exercise

Convert some semantic patch that you have previously written so that it prints an error message rather than making a match or change.

Detecting memory leaks

A simple case of a memory leak:

- An allocation.
- Storage in a local variable.
- No use.
- Return of an error code (negative constant).

Example:

```
tmp_store = kmalloc(sizeof(*tmp_store), GFP_KERNEL);
if (!tmp_store) {
  ti->error = "Exception store allocation failed";
  return -ENOMEM;
}

persistent = toupper(*argv[1]);
if (persistent != 'P' && persistent != 'N') {
  ti->error = "Persistent flag is not P or N";
  return -EINVAL;
}
```

Exercise

Write a semantic match that detects memory leaks involving `kmalloc`, `kcalloc`, or `kzalloc`.