Isomorphisms for the Coccinelle Program Matching and Transformation Engine

Julia Lawall (University of Copenhagen)

Joint work with
Jesper Andersen, Julien Brunel, Damien Doligez,
René Rydhof Hansen, Bjørn Haagensen,
Gilles Muller, Yoann Padioleau, and Nicolas Palix
DIKU-Aalborg-EMN

June 2009

Overview

Goal: Describe and automate transformations on C code

1. Collateral evolutions.
2. Bug finding and fixing.

- Focus on open-source software, particularly Linux.

Our approach: Coccinelle

- Semantic patch language (SmPL).

Isomorphisms.

- Projecting transformations onto “isomorphic” terms.
- Example: *x* `==` `NULL` vs. `NULL` `==` *x*.

Conclusions and future work.

Collateral evolutions

The collateral evolution problem:

- Library functions change.
- Client code must be adapted.
  - Change a function name, add an argument, etc.
- Linux context:
  - Many libraries: usb, net, etc.
  - Very many clients, including outside the Linux source tree.

Example

Evolution: New constants:

`IRQF_DISABLED`, `IRQF_SAMPLE_RANDOM`, *etc.*

$\Longrightarrow$ Collateral evolution: Replace old constants by the new ones.

```
@@ -96,7 +96,7 @@ static int __init hp6x0_apm_init(void)
   int ret;

   ret = request_irq(HP680_BTN_IRQ, hp6x0_apm_interrupt,
-                    SA_INTERRUPT, MODNAME, 0);
+                    IRQF_DISABLED, MODNAME, 0);
   if (unlikely(ret < 0)) {
           printk(KERN_ERR MODNAME ": IRQ %d request failed",
                  HP680_BTN_IRQ);
```

Changes required in 547 files, over 3 months

Bug finding and fixing

Bad combination of boolean and bit operators

- `!` always returns 1 or 0
- `CENTER_LFE_ON` is 0x0020

```
if (!state->card->
    ac97_status & CENTER_LFE_ON)
  val &= ~DSP_BIND_CENTER_LFE;
```

A more complex collateral evolution

Evolution: A new function: kzalloc

⟹ Collateral evolution: Merge kmalloc and memset into kzalloc

```
fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (fh == NULL) {
  dprintk(1,
    KERN_ERR
    "%s: zoran_open(): allocation of zoran_fh failed\n",
    ZR_DEVNAME(zr));
  return -ENOMEM;
}
memset(fh, 0, sizeof(struct zoran_fh));
```

A more complex collateral evolution

Evolution: A new function: kzalloc
⟹ Collateral evolution: Merge kmalloc and memset into kzalloc

```
fh = kzalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (fh == NULL) {
  dprintk(1,
    KERN_ERR
    "%s: zoran_open(): allocation of zoran_fh failed\n",
    ZR_DEVNAME(zr));
  return -ENOMEM;
}
```

Existing tools

Collateral evolutions

- Refactoring tools in various IDEs
- Typically restricted to a fixed set of semantics-preserving transformations
- Typically require the availability of all source code

Bug finding

- Metal/Coverity, SLAM/SDV, Splint, Flawfinder, etc.
- Limited user control - in practice often used as a black box.
- No support for bug fixing.

Our proposal: Coccinelle

Program matching and transformation for unpreprocessed C code.

Semantic Patches:

- Like patches, but independent of irrelevant details (line numbers, spacing, variable names, etc.)
- Derived from code, with abstraction.
- Goal: fit with the existing habits of the Linux programmer.

Example: SA/IRQF collateral evolution

```
@@ @@
(
- SA_INTERRUPT
+ IRQF_DISABLED
|
- SA_SAMPLE_RANDOM
+ IRQF_SAMPLE_RANDOM
|
- SA_SHIRQ
+ IRQF_SHARED
|
- SA_PROBEIRQ
+ IRQF_PROBE_SHARED
|
- SA_PERCPU_IRQ
+ IRQF_PERCPU
)
```

Example: boolean/bit bug finding and fixing

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

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

Constructing a semantic patch

```
fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (fh == NULL) {
  dprintk(1,
    KERN_ERR
    "%s: zoran_open(): allocation of zoran_fh failed\n",
    ZR_DEVNAME(zr));
  return -ENOMEM;
}
memset(fh, 0, sizeof(struct zoran_fh));
```

Constructing a semantic patch

Eliminate irrelevant code

```
fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
if (fh == NULL) {


  ...

  return ...;
}
memset(fh, 0, sizeof(struct zoran_fh));
```

Constructing a semantic patch

Describe transformations

```
- fh = kmalloc(sizeof(struct zoran_fh), GFP_KERNEL);
+ fh = kzalloc(sizeof(struct zoran_fh), GFP_KERNEL);
  if (fh == NULL) {
    ...
    return ...;
  }
- memset(fh, 0, sizeof(struct zoran_fh));
```

Constructing a semantic patch

Abstract over subterms

```
@@
expression x,E1,E2;
@@

- x = kmalloc(E1,E2);
+ x = kzalloc(E1,E2);
  if (fh == NULL) {
    ...
    return ...;
  }
- memset(x, 0, E1);
```

Practical results

Collateral evolutions

- Semantic patches for over 60 collateral evolutions.
- Applied to over 5800 Linux files from various versions, with a success rate of 100% on 93% of the files.

Bug finding

- Generic bug types:
  - Null pointer dereference, initialization of unused variables, `!x&y`, etc.
- Bugs in the use of Linux APIs:
  - Incoherent error checking, memory leaks, etc.

Over 280 patches created using Coccinelle accepted into Linux

Starting to be used by other developers of C code

Probable bugs found in gcc, postgresql, vim, amsn, pidgin, mplayer

But wait...

```
@@
expression x,E1,E2;
@@

- x = kmalloc(E1,E2);
+ x = kzalloc(E1,E2);
  if (x == NULL) {
    ...
    return ...;
  }
- memset(x,0,E1);
```

updates 38/564 files

Issues

```
@@
expression x,E1,E2;
@@

- x = kmalloc(E1,E2);
+ x = kzalloc(E1,E2);
  if (x == NULL) {
    ...
    return ...;
  }
- memset(x,0,E1);
```

- Some code uses `!x` or `NULL == x`.
- Some code has only the return in the error handling code.
  - Linux code doesn't use `{}` around a single statement branch.
- Some code uses `return;`

Isomorphisms to the rescue

```
Expression
@ is_null @
expression X;
@@
X == NULL <=> NULL == X => !X

Statement
@ braces1 @
statement S;
@@
{ ... S } => S

Statement
@ ret @
@@
return ...; => return;
```

Example

```
@@
expression x,E1,E2;
@@

- x = kmalloc(E1,E2);
+ x = kzalloc(E1,E2);
  if (x == NULL) {
    ...
    return ...;
  }
- memset(x,0,E1);
```

Now matches the Linux code (zfcp_scsi.c):

```
data = kmalloc(sizeof(*data), GFP_KERNEL);
if (!data)
  return;
memset(data, 0, sizeof(*data));
```

updates 205/564 files

Are isomorphisms always safe to apply?

```
Expression
@ is_null_simplified @
expression X;
@@
X == NULL => !X
```

Consider the semantic patch:

```
@ bad_patch @
expression A;
@@
  A ==
-       NULL
+       7
```

- The transformation becomes `( A == NULL` $^{-+7}$ `| !A )`
- Oops!

Are isomorphisms always safe to apply?

```
Expression
@ is_null_simplified @
expression X;
@@
X == NULL => !X
```

Consider the semantic patch:

```
@ good_patch @
expression A;
@@
- A == NULL
+ A == 7
```

- The transformation becomes `( A == NULL | !A )` $^{-+\texttt{A == 7}}$
- OK, but the coding style is not preserved.

Are isomorphisms always safe to apply?

```
Expression
@ is_null_simplified @
expression X;
@@
X == NULL => !X
```

Consider the semantic patch:

```
@ another_good_patch @ expression A; @@
- A
+ 7
    == NULL
```

- The transformation becomes (A^{-+7} == NULL | !A^{-+7})
- OK. Coding style also preserved.

Rules for safe isomorphisms

- An isomorphism can match a completely - pattern.
- Otherwise, only an isomorphism metavariable can match a pattern containing a transformation.
- ...

Are isomorphisms always safe to apply?

```
Expression
@ bad_double_iso @
expression X;
@@
X * 2 => X + X
```

The semantic patch:

```
@ double_bc @ @@
( b | c ) * 2
```

Becomes:

```
@ bad_double_iso_double_bc @ @@
(   ( b | c ) * 2   |   ( b | c ) + ( b | c )   )
```

Oops, again...

Rules for safe isomorphisms

- An isomorphism can match a completely removed pattern.
- Otherwise, only an isomorphism metavariable can match a pattern containing a transformation.
- Isomorphism metavariables that are duplicated on the right-hand side cannot match disjunctions.
- Something else?

Correctness constraint

$$
\begin{array}{l}
\text{correct}(g) \Leftrightarrow \\
\quad \forall \rho \in \text{environments} : \\
\quad\quad \forall C \in \text{contexts} : \\
\quad\quad\quad \forall f \in \text{semantic patches} : \\
\quad\quad\quad\quad g \sim_{\rho,C} f \Rightarrow \\
\quad\quad\quad\quad\quad \forall \sigma \in \text{environments} : \\
\quad\quad\quad\quad\quad\quad \forall \tau \in \text{traces} : \\
\quad\quad\quad\quad\quad\quad\quad \forall E \in \text{programs} : \\
\quad\quad\quad\quad\quad\quad\quad\quad g(\rho, C, f) \sim_{\sigma,\tau} E \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \exists \sigma' \in \text{environments} : \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \exists \tau' \in \text{traces} : \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \exists E' \in \text{programs} : \\
f \sim_{\sigma',\tau'} E' \wedge \sigma \| \sigma' \wedge [\![E]\!] = [\![E']\!] \wedge [\![(g(\rho, C, f))(\sigma, \tau, E)]\!] = [\![f(\sigma', \tau', E')]\!]
\end{array}
$$

- If an isomorphism g matches a semantic patch f, and
- If the result of applying g to f matches the code E,
- Then, there should be some term E' that would have been matched by f such that:
  - E and E' have the same semantics.
  - The transformed versions of E and E' have the same semantics.

Reasonableness constraint

The correctness constraint requires thinking at two levels...

$$
\begin{array}{l}
\text{reasonable}(I_1 \;\texttt{=>}\; I_2) \Leftrightarrow \\
\quad \forall \sigma \in \text{environments} : \\
\quad\quad \forall \tau \in \text{traces} : \\
\quad\quad\quad \forall E \in \text{programs} : \\
\quad\quad\quad\quad I_2 \sim_{\sigma,\tau} E \Rightarrow \\
\quad\quad\quad\quad\quad\quad \exists \sigma' \in \text{environments} : \\
\quad\quad\quad\quad\quad\quad\quad \exists \tau' \in \text{traces} : \\
\quad\quad\quad\quad\quad\quad\quad\quad \exists E' \in \text{programs} : \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad I_1 \sim_{\sigma',\tau'} E' \wedge \sigma \| \sigma' \wedge [\![E]\!] = [\![E']\!]
\end{array}
$$

- If I_2 matches a term E, then
- There should be some term E' such that
  - I_1 matches E'
  - E and E' have the same semantics.

Future work

Does reasonable(g) $\Rightarrow$ correct(g)???

- Probably not...

Or perhaps reasonable(g) $\wedge\ \phi \Rightarrow$ correct(g), for some ϕ???

Stay tuned...

Conclusion

A patch-like program matching and transformation language

Converting this notation into an implementation raises some issues:

- Extension to CTL for matching control-flow paths.
- Isomorphisms for simplifying the manually written patterns.

Over 280 patches created using Coccinelle accepted into Linux

Future work

- Put the isomorphism idea on firmer foundations.
- Consider programming languages other than C.
- Integrate dataflow and interprocedural analysis.

Coccinelle is publicly available

`http://www.emn.fr/x-info/coccinelle/`