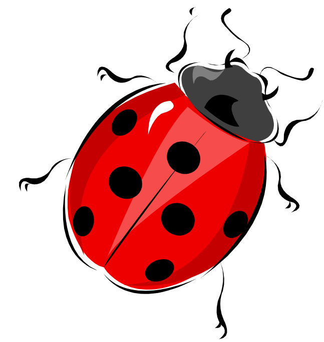
# Semantic Patches
## for specifying and automating
# Collateral Evolutions




Yoann Padioleau

Ecole des Mines de Nantes, France

with

René Rydhof Hansen and Julia Lawall (DIKU, Denmark)

Gilles Muller (Ecole des Mines de Nantes)


the Coccinelle project

« The Linux USB code has been rewritten at least three times. We've done this over time in order to handle things that we didn't originally need to handle, like high speed devices, and just because we learned the problems of our first design, and to fix bugs and security issues. Each time we made changes in our API, we updated all of the kernel drivers that used the APIs, so nothing would break. And we deleted the old functions as they were no longer needed, and did things wrong. »

- Greg Kroah-Hartman, OLS 2006.

# The problem: Collateral Evolutions

Evolution in a library

lib.c

```
int foo(int x) {
```

becomes

```
int bar(int x) {
```

Legend:

before

after

Can entail lots of Collateral Evolutions in clients

client1.c

```
foo(1);
```

```
bar(1);
```

```
foo(2);
```

```
bar(2);
```

client2.c

```
foo(foo(2));
```

```
bar(bar(2));
```

```
if(foo(3))  {
```

```
if(bar(3))  {
```

clientn.c

# Our main target: device drivers

Many libraries: driver support libraries

- One per device type, per bus (pci library, sound, …)

Many clients: device specific code

- Drivers make up > 50% of the Linux source code

Many evolutions and collateral evolutions

- 1200 evolutions in 2.6, some affecting 400 files, at over 1000 sites

Taxonomy of evolutions :

- Add argument, split data structure, getter and setter introduction, change protocol sequencing, change return type, add error checking, …

# Our goal

- Currently, Collateral Evolutions in Linux are done nearly manually:
  - Difficult
  - Time consuming
  - Error prone
- The highly concurrent and distributed nature of the Linux development process makes it even worse:
  - Misunderstandings
  - Out of date patches, conflicting patches
  - Patches that miss code sites (because newly introduced sites and newly introduced drivers)

Need a tool to document and automate Collateral Evolutions

# Complex Collateral Evolutions

The *proc_info* functions should not call the **scsi_get** and **scsi_put** library functions to compute a scsi resource. This resource will now be passed directly to those functions via a parameter.

```
int proc_info(int x
                  ,scsi *y
                        ) {
   scsi *y;
   ...
   y = scsi_get();
   if(!y) { ... return -1; }
   ...
   scsi_put(y);
   ...
}
```

From local var to parameter

Delete calls to library

Delete error checking code

# Excerpt of patch file

```
@@    -246,7 +246,8@@
  - int wd7000_info(int a) {
  + int wd7000_info(int a,scsi b) {
    int z;
  - scsi *b;
    z = a + 1;
  - b = scsi_get();
  - if(!b) {
  -    kprintf("error");
  -    return -1;
  - }
    kprintf("val = %d", b->field + z);
  - scsi_put(b);
    return 0;
   }
```

- Similar (but not identical) transformation done in other drivers
- A patch is specific to a file, to a code site
- A patch is line-oriented

# Our idea

## The example

```
int proc_info(int x
                    ,scsi *y
                           ) {
   scsi *y;
   ...
   y = scsi_get();
   if(!y) { ... return -1; }
   ...
   scsi_put(y);
   ...
}
```

How to specify the required program transformation ?

In what programming language ?

# Our idea: Semantic Patches

```
@@
  function proc_info;
  identifier x,y;
@@
     int proc_info(int x
+                   ,scsi *y
                       ) {
-      scsi *y;
       ...
-      y = scsi_get();
-      if(!y) { ... return -1; }
       ...
-      scsi_put(y);
       ...
     }
```

metavariables

Declarative language

the '...' operator

modifiers

# SmPL: Semantic Patch Language

- A single small semantic patch can modify hundreds of files, at thousands of code sites
  - before: **$ patch -p1 < wd7000.patch**
  - now: **$ spatch *.c < proc_info.spatch**

- The features of SmPL make a semantic patch generic by abstracting away the specific details and variations at each code site among all drivers:
  - Differences in spacing, indentation, and comments
  - Choice of names given to variables (use of metavariables)
  - Irrelevant code (use of '...' operator)
  - Other variations in coding style (use of isomorphisms)
    - e.g. if(!y) ≡ if(y==NULL) ≡ if(NULL==y)

# The full semantic patch

```
@ rule1 @
struct SHT fops;
identifier proc_info;
@@
 fops.proc_info = proc_info;

@ rule2 @
identifier rule1.proc_info;
identifier buffer, start, inout, hostno;
identifier hostptr;
@@
  proc_info (
+       struct Scsi_Host *hostptr,
        char *buffer, char **start,
-       int hostno,
        int inout) {
    ...
-   struct Scsi_Host *hostptr;
    ...
-   hostptr = scsi_host_hn_get(hostno);
    ...
?-  if (!hostptr) { ... return ...; }
    ...
?-  scsi_host_put(hostptr);
    ...
  }
```

```
@ rule3 @
identifier rule1.proc_info;
identifier rule2.hostno;
identifier rule2.hostptr;
@@
  proc_info(...) {
    <...
-   hostno
+   hostptr->host_no
    ...>
  }
@ rule4 @
identifier rule1.proc_info;
identifier func;
expression buffer, start, inout, hostno;
identifier hostptr;
@@

 func(..., struct Scsi_Host *hostptr, ...) {
  <...
   proc_info(
+        hostptr,
         buffer, start,
-        hostno,
         inout)
   ...>
 }
```

# SmPL piece by piece

# Concrete code & modifiers (1/2)

```
  proc_info(
+       struct Scsi_Host *hostptr,
        char *buf, char **start,
-       int hostno,
        int inout) {
```

```
-   proc_info(char *buf, char **start,
-           int hostno, int inout)
+   proc_info(struct Scsi_host *hostptr,
+           char *buf, char **start, int inout)
{
```

Can write almost any C code, even some CPP directives

Can annotate with +/- almost freely

Can often start a semantic patch by copy pasting from a regular patch (and then generalizing it)

Can update prototypes automatically (in .c or .h)

# Concrete code & modifiers (2/2)

Some examples:

```
@@
expression E; type T;
@@
  E =
-    (T)
     kmalloc(...)
```

```
@@
expression N;
@@
- N & (N-1)
+ is_power_of_2(N)
```

```
@@
expression X;
@@
- memset(X,0, PAGE_SIZE)
+ clear_page(X)
```

Simpler than regexps:

```
perl -pi -e "s/ ?= ?\([^\)]*\) *(kmalloc) *\(/ = \1\(/"
grep -e "([^\(\)]+) ?\& ?\(\1 ?- ?1\)"
grep -e "memset ?\([^,]+, ?, ?0, ?PAGE_SIZE\) "
```

Insensitive to differences in spaces, newlines, comments

# Metavariables and the rule

```
@@
identifier proc_info;
identifier buffer, start,inout, hostno;
identifier hostptr;
@@
  proc_info (
+       struct Scsi_Host *hostptr,
        char *buffer, char **start,
-       int hostno,
        int inout) {
     ...
-    struct Scsi_Host *hostptr;
     ...
-    hostptr = scsi_host_hn_get(hostno);
     ...
-    if (!hostptr) { ... return ...; }
     ...
-    scsi_host_put(hostptr);
     ...
  }
```

Metavariables:

- Abstract away names given to variables
- Store "values"
- Constrain the transformation when a metavariable is used more than once
- Can be used to move code

Search in whole file

Match, bind, transform

Transform only if everything matches

Can match/transform

metavariables declaration + code patterns = a rule

# Multiples rules and inherited metavariables

```
@ rule1 @
struct SHT fops;
identifier proc_info;
@@
 fops.proc_info = proc_info;

@ rule2 @
identifier rule1.proc_info_func;
identifier buf, start, inout, hostno;
identifier hostptr;
@@
  proc_info (
+       struct Scsi_Host *hostptr,
        char *buf, char **start,
-       int hostno,
```

- Each rule matched against the whole file
- Can communicate information/constraints between rules
- Anonymous rules vs named rules
- Inherited metavariables
- Can move code between functions

Note, some rule don't contain transformation at all

Can have typed metavariable

# Sequences and the '...' operator (1/2)

Source code

1

```
b = scsi_get();
if(!b) return -1;
kprintf("val = %d", b->field + z);
scsi_put(b);
return 0;
```

2

```
sc = scsi_get();
if(!sc) { kprintf("err"); return -1;}
if(y<2) {
   scsi_put(sc);
   return -1;
}
kprintf("val = %d", sc->field + z);
scsi_put(sc);
return 0;
```

3

```
b = scsi_get();
if(!b) return -1;
switch(x) {
  case V1: i++; scsi_put(b); return i;
  case V2: j++; scsi_put(b); return j;
  default:
   scsi_put(b);
   return 0;
}
```

Some running execution

| D1 | D2 | D3 |
|---|---|---|
| `scsi_get()` | `scsi_get()` | `scsi_get()` |
| ... | ... | ... |
| `scsi_put()` | `scsi_put()` | `scsi_put()` |

time

Always one **scsi_get** and one **scsi_put** per execution

Syntax differs but executions follow same pattern

# Sequences and the '...' operator (2/2)

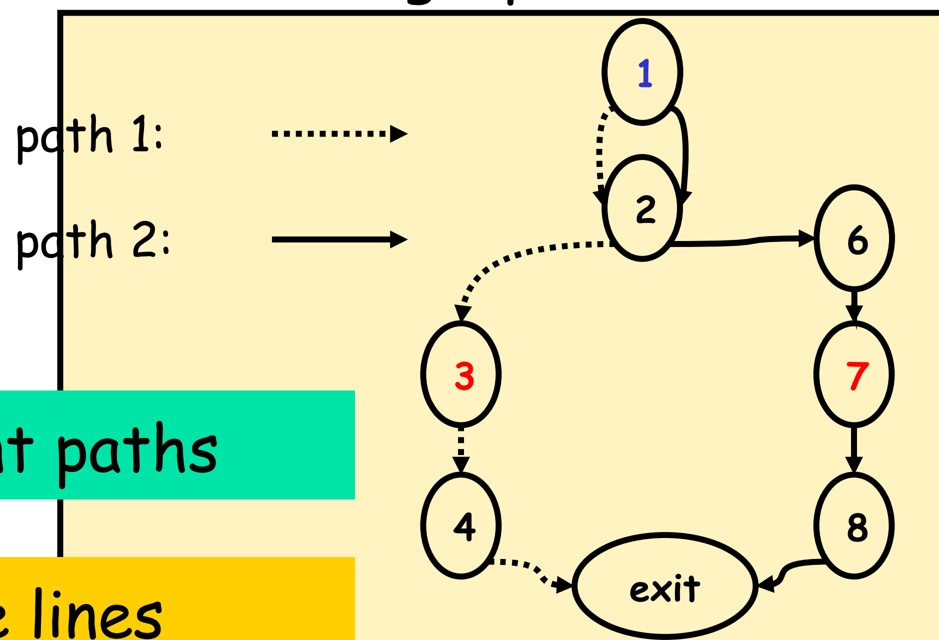
C file

```
1 y = scsi_get();
2 if(exp) {
3  scsi_put(y);
4  return -1;
5 }
6 printf("%d",y->f);
7 scsi_put(y);
8 return 0;
```

Semantic patch

```
- y = scsi_get();
  ...
- scsi_put(y);
```

Control-flow graph of C file




"..." means for all subsequent paths

One '-' line can erase multiple lines

# Isomorphisms (1/2)

Examples:

- Boolean : X == NULL ⇔ !X ⇔ NULL == X
- Control : if(E) S1 else S2 ⇔ if(!E) S2 else S1
- Pointer : E->field ⇔ *E.field
- etc.

```
@@ expression *X; @@
X == NULL   <=>   !X   <=>   NULL == X
```

We have reused SmPL syntax

# Isomorphisms (2/2)

```
@ rule1 @
struct SHT fops;
identifier proc_info;
@@
 fops.proc_info = proc_info;
```

1
```
myops->proc_info = scsiglue_info;
myops->open = scsiglue_open;
```

2
```
struct SHT wd7000 = {
   .proc_info = wd7000_proc_info,
   .open = wd7000_open,
}
```

```
   ...
- if (!hostptr) { ... return...; }
   ...
```

3
```
if(!hostptr == NULL)
   return -1;
```

standard isos

```
@@
type T;
T E, *E1;
identifier fld;
@@
E.fld <=> E1->fld

@@
type T; T E;
identifier v, fld;
expression E1;
@@
E.fld = E1; => T v = { .fld = E1, };

@@ expression *X; @@
X == NULL <=> NULL == X <=> !X

@@ statement S; @@
{ ... S ... } => S
```

# Nested sequences

```
@ rule3 @
identifier rule1.proc_info;
identifier rule2.hostno;
identifier rule2.hostptr;
@@
  proc_info(...) {
    <...
-   hostno
+   hostptr->host_no
    ...>
  }
```

An execution in one driver

```
enter proc_info
      ...
access hostno
      ...
modify hostno
      ...
access hostno
      ...
exit proc_info
```

time

Global substitution (a la /g) but with delimited scope

For full global substitution do:

```
@@ @@
-   hostno
+   hostptr->host_no
```

# The full semantic patch

```
@ rule1 @
struct SHT fops;
identifier proc_info;
@@
 fops.proc_info = proc_info;

@ rule2 @
identifier rule1.proc_info;
identifier buffer, start, inout, hostno;
identifier hostptr;
@@
  proc_info (
+      struct Scsi_Host *hostptr,
       char *buffer, char **start,
-      int hostno,
       int inout) {
    ...
-   struct Scsi_Host *hostptr;
    ...
-   hostptr = scsi_host_hn_get(hostno);
    ...
?-  if (!hostptr) { ... return ...; }
    ...
?-  scsi_host_put(hostptr);
    ...
  }
```

```
@ rule3 @
identifier rule1.proc_info;
identifier rule2.hostno;
identifier rule2.hostptr;
@@
  proc_info(...) {
    <...
-   hostno
+   hostptr->host_no
    ...>
  }

@ rule4 @
identifier rule1.proc_info;
identifier func;
expression buffer, start, inout, hostno;
identifier hostptr;
@@

 func(..., struct Scsi_Host *hostptr, ...) {
  <...
   proc_info(
+        hostptr,
         buffer, start,
-        hostno,
         inout)
   ...>
 }
```

# More examples

# More examples: video_usercopy

C file

```
int p20_ioctl(int cmd, void*arg)
 switch(cmd) {
  case VIDIOGCTUNER: {
    struct video_tuner v;
    if(copy_from_user(&v,arg)!=0)
       return -EFAULT;
    if(v.tuner)
       return -EINVAL;
    v.rangelow = 87*16000;
    v.rangehigh = 108 * 16000;
    if(copy_to_user(arg,&v))
       return -EFAULT;
    return 0;
  }
  case AGCTUNER: {
     struct video_tuner v;
```

Semantic Patch

```
@@     type T;identifier x,fld;@@
  ioctl(...,void *arg,...) {
  <...
- T x;
+ T *x = arg;
   ...
- if(copy_from_user(&x, arg))
-   { ... return ...; }
    <...
(
-     x.fld
+     x->fld
|
-      &x
+      x
)
    ...>
- if(copy_to_user(arg,&x))
- { ... return ... }
 ...>
}
```

Nested pattern

Iso

Iso

Disjunction pattern

Nested end pattern

# More examples: video_usercopy

C file

```
int p20_ioctl(int cmd, void*arg)
 switch(cmd) {
  case VIDIOGCTUNER: {
    struct video_tuner *v = arg;


    if(v->tuner)
       return -EINVAL;
    v->rangelow = 87*16000;
    v->rangehigh = 108 * 16000;


    return 0;
  }
  case AGCTUNER: {
     struct video_tuner *v = arg;
```

Semantic Patch

```
@@     type T;identifier x,fld;@@
  ioctl(...,void *arg,...) {
  <...
- T x;
+ T *x = arg;
   ...
- if(copy_from_user(&x, arg))
-   { ... return ...; }
    <...
(
-       x.fld
+       x->fld
|
-        &x
+        x
)
    ...>
- if(copy_to_user(arg,&x))
- { ... return ... }
 ...>
}
```

Nested pattern

Iso

Iso

Disjunction pattern

Nested end pattern

# More examples: check_region

C file

```
if(check_region(piix,8)){
   printk("error1");
   return -ENODEV;
}
if(force_addr) {
   printk("warning1");
} else if((temp & 1) == 0) {
   if(force) {
     printk("warning2");
   } else {
     printk("error2");

     return -ENODEV;
   }
}
request_region(piix,8);
printk("done");
```

Semantic Patch

```
@@     expression e1,e2;@@
- if(check_region(e1,e2)!=0)
+ if(!request_region(e1,e2))
  { ... return ... }
  <...
+ release_region(e1)
  return ...;
  ...>
- request_region(e1,e2);
```

# More examples: check_region

C file

```
if(!request_region(piix,8)){
   printk("error1");
   return -ENODEV;
}
if(force_addr) {
   printk("warning1");
} else if((temp & 1) == 0) {
   if(force) {
     printk("warning2");
   } else {
     printk("error2");
     release_region(piix);
     return -ENODEV;
   }
}

printk("done");
```

Semantic Patch

```
@@    expression e1,e2;@@
- if(check_region(e1,e2)!=0)
+ if(!request_region(e1,e2))
  { ... return ... }
  <...
+ release_region(e1)
  return ...;
  ...>
- request_region(e1,e2);
```

# How does it work ?

This is pure magic ™

# Our vision

- The library maintainer performing the evolution also writes the semantic patch (SP) that will perform the collateral evolutions
- He looks a few drivers, writes SP, applies it, refines it based on feedback from our interactive engine, and finally sends his SP to Linus
- Linus applies it to the lastest version of Linux, to the newly added code sites and drivers
- Linus puts the SP in the SP repository so that device drivers outside the kernel can also be updated

# Conclusion

- Collateral Evolution is an important problem, especially in Linux device drivers
- SmPL: a declarative language to specify collateral evolutions
- Looks like a patch; fits with Linux programmers' habits
- But takes into account the semantics of C (execution-oriented, isomorphisms), hence the name Semantic Patches
- A transformation engine to automate collateral evolutions. Our tool can be seen as an advanced refactoring tool for the Linux

# Your opinion

- We would like your opinion
  - Nice language ? Too complex ?
  - Collateral evolutions are not a problem for you ?
  - Ideas to improve SmPL ?
  - Examples of evolutions/collateral evolutions you would like to do ?
  - Would you like to collaborate with us and try our tool ?
- Any questions ? Feedback ?
- Contact: **padator@wanadoo.fr**