Semantic Patches

Documenting and Automating Collateral Evolutions in Linux Device Drivers

Yoann Padioleau
EMN
padator@wanadoo.fr

Julia L. Lawall
DIKU
julia@diku.dk

Gilles Muller
EMN
Gilles.Muller@emn.fr

1 Introduction

Device drivers form the glue code between an operating system and its devices. In Linux, device drivers are highly reliant for this on the various Linux internal libraries, which encapsulate generic functionalities related to the various busses and device types. In recent years, these libraries have been evolving rapidly, to address new requirements and improve performance. In response to each evolution, *collateral evolutions* are often required in driver code, to bring the drivers up to date with the new library API. Currently, collateral evolutions are mostly done manually. The large number of drivers, however, implies that this approach is time-consuming and unreliable, leading to subtle errors when modifications are not done consistently.

To address this problem, we propose a scripting language for specifying and automating collateral evolutions. This language offers a WYSIWYG approach to program transformation. In the spirit of Linux development practice, this language is based on the patch syntax. As opposed to traditional patches, our patches are not line-oriented but semantics-oriented, and hence we give them the name *semantic patches*.

This paper gives a tutorial on our semantic patch language, SmPL, and its associated transformation tool, `spatch`. We first give an idea of the kind of program transformations we target, collateral evolutions, and then present SmPL using an example based on Linux driver code. We then present some further examples of evolutions and collateral evolutions that illustrate other issues in semantic patch development. Finally, we describe the current status of our project and propose some future work. Our work is directed mainly to device driver maintainers, library developers, and kernel janitors, but anyone who has ever performed a repetitive editing task on C code can benefit from it.

2 Evolutions and Collateral Evolutions

The evolutions we consider are those that affect a library API. Elements of a library API that can be affected include functions, both those defined by the library and the callback functions that the library expects to receive from a driver, global variables and constants, types, and macros. A library may also implicitly specify rules for using these elements. Many kinds of changes in the API can result from an evolution that affect one of these elements. For example, functions or macros can change name or gain or lose arguments. Structure types can be reorganized and accesses to them can be encapsulated in getter and setter functions. The protocol for using a sequence of functions, such as `up` and `down` can change, as can the protocol for when error checking is needed and what kind of error values should be returned.

Each of these changes requires corresponding collateral evolutions, in all drivers using the API. When a function or macro changes name, all callers need to be updated with the new name. When a function or macro gains or loses arguments, new argument values have to be constructed and old ones have to be dropped in the driver code, respectively. When a structure type is reorganized, all drivers accessing affected fields of that structure have to be updated, either to perform the new field references or to use any introduced getter and setter functions. Changes in protocols may require a whole sequence of modifications, to remove the old code and introduce the new. Many of these collateral evolutions can have a non-local effect, as, for example, changing a new argument value may trigger a whole set of new computations, and changing a protocol may require substantial code restructuring. The interaction of a driver with the API may furthermore include some device-specific aspects. Thus, these changes have to be mapped onto the structure of each affected driver file.

We have characterized evoluations and collateral evolu-

tions in more detail, including numerous examples, in a paper at EuroSys 2006 [1].

3 Semantic Patch Tutorial

In this section, we describe SmPL (Semantic Patch Language), our language for writing semantic patches. To motivate the features of SmPL, we first consider a moderately complex collateral evolution that raises many typical issues. We then present SmPL in terms of this example.

3.1 The "proc_info" evolution

As an example, we consider an evolution and associated collateral evolutions affecting the SCSI API functions `scsi_host_hn_get` and `scsi_host_put`. These functions access and release, respectively, a structure of type `Scsi_Host`, and additionally increment and decrement, respectively, a reference count. In Linux 2.5.71, it was decided that, due to the criticality of the reference count, driver code could not be trusted to use these functions correctly and they were removed from the SCSI API [2].

This evolution had collateral effects on the "proc_info" callback functions defined by SCSI drivers, which call these API functions. Figure 1 shows a slightly simplified excerpt of the traditional patch file updating the proc_info function of `drivers/usb/storage/scsiglue.c`. Similar collateral evolutions were performed in Linux 2.5.71 in 18 other SCSI driver files inside the kernel source tree. To compensate for the removal of `scsi_host_hn_get` and `scsi_host_put`, the SCSI library began in Linux 2.5.71 to pass to these callback functions a `Scsi_Host`-typed structure as an argument. Collateral evolutions were then needed in all the proc_info functions to remove the calls to `scsi_host_hn_get` (line 19 for the `scsiglue.c` driver), and `scsi_host_put` (lines 27 and 42), and to add the new argument (line 4). Those changes in turn entailed the removal of a local variable (line 11) and of null-checking code (line 20-22), as the library is assumed not to call the proc_info function on a null value. Finally, one of the parameters of the proc_info function was dropped (line 6) and every use of this parameter was replaced by a field access (line 33) on the new structure argument.

```
0  --- a/drivers/usb/storage/scsiglue.c
1  +++ b/drivers/usb/storage/scsiglue.c
2  @@ -264,33 +300,21 @@
3  -static int usb_storage_proc_info (
4  +static int usb_storage_proc_info (struct Scsi_Host *hostptr,
5                 char *buffer, char **start, off_t offset,
6  -              int hostno, int inout)
7  +              int inout)
8   {
9     struct us_data *us;
10    char *pos = buffer;
11 -  struct Scsi_Host *hostptr;
12    unsigned long f;
13
14    /* if someone is sending us data, just throw it away */
15    if (inout)
16            return offset;
17
18 -  /* find our data from the given hostno */
19 -  hostptr = scsi_host_hn_get(hostno);
20 -  if (!hostptr) {
21 -          return -ESRCH;
22 -  }
23    us = (struct us_data*)hostptr->hostdata[0];
24
25    /* if we couldn't find it, we return an error */
26    if (!us) {
27 -          scsi_host_put(hostptr);
28            return -ESRCH;
29    }
30
31    /* print the controller name */
32 -  SPRINTF("   Host scsi%d: usb-storage\n", hostno);
33 +  SPRINTF("   Host scsi%d: usb-storage\n", hostptr->host_no);
34    /* print product, vendor, and serial number strings */
35    SPRINTF("       Vendor: %s\n", us->vendor);
36
37 @@ -318,9 +342,6 @@
38      *(pos++) = '\n';
39     }
40
41 -  /* release the reference count on this host */
42 -  scsi_host_put(hostptr);
43
44    /*
45     * Calculate start of next buffer, and return value.
46
```

Figure 1: Simplified excerpt of the patch file from Linux 2.5.70 to Linux 2.5.71

Of the possible API changes identified in Section 2, this example illustrates the dropping of two library functions and changes in the parameter list of a callback function. These changes have non-local effects in the driver code, as the context of the dropped call to `scsi_host_hn_get` must change as well, to eliminate the storage of the result and the subsequent error check, and the value of the dropped parameter must be reconstructed wherever it is used.

3.2 A semantic patch, step by step

We now describe the semantic patch that will perform the previous collateral evolutions, on any of the 19 relevant files inside the kernel source tree, and on any relevant drivers outside the kernel source tree. We first describe step-by-step various excerpts of this semantic

patch, and then present its complete definition in Section 3.3.

3.2.1 Modifiers

The first excerpt adds and removes the affected parameters of the proc_info callback function:

```
  proc_info_func (
+     struct Scsi_Host *hostptr,
      char *buffer, char **start, off_t offset,
-     int hostno,
      int inout) { ... }
```

Like a traditional patch, a semantic patch consists of a sequence of lines, some of which begin with the *modifiers* + and − in the first column. These lines are added or removed, respectively. The remaining lines serve as *context*, to more precisely identify the code that should be modified.

Unlike a traditional patch, a semantic patch must have the form of a complete C-language term (an expression, a statement, a function definition, etc.). Here we are modifying a function, so the semantic patch has the form of a function definition. Because our only goal at this point is to modify the parameter list, we do not care about the function body. Thus, we have represented it with "...". The meaning and use of "..." are described in more detail in Section 3.2.4.

Because of the wide range of possible collateral evolutions, as described in Section 2, collateral evolutions may affect almost any C constructs, such as structures, initializers, function parameters, `if` statements, in many different ways. SmPL, for flexibility, allows to write almost any C code in a semantic patch and to annotate freely any part of this code with the + and − modifiers. The combination of the unannotated context code with the − code and the combination of the unannotated context code with the + code must, however, each have the form of valid C code, to ensure that the pattern described by the former can match against valid driver code and that the transformation described by the latter will produce valid C code as a result.

Another difference as compared to a traditional patch is that the meaning of a semantic patch is insensitive to newlines, spaces, comments, etc. Thus, the above semantic patch will *match* and *transform* driver code that has the parameters of the proc_info function all on the same line, spread over multiple lines as in `scsiglue.c`, or separated by comments. We have split the semantic patch over four lines only to better highlight what is added and removed. We could have equivalently written it as:

```
- proc_info_func(char *buffer, char **start, off_t offset,
-                int hostno, int inout)
+ proc_info_func(struct Scsi_Host *hostptr, char *buffer,
+                char **start, off_t offset, int inout)

    { ... }
```

To apply this semantic patch, it should be stored in a file, *e.g.*, `procinfo.spatch`. It can then be applied to *e.g.* the set of C files in the current directory using our `spatch` tool:

```
spatch *.c < procinfo.spatch.
```

3.2.2 Metavariables

A traditional patch, like the one in Figure 1, describes a transformation of a specific set of lines in a specific driver. This specificity is due to the fact that a patch hardcodes some information, such as the name of the driver's proc_info callback function. Thus a separate patch is typically needed for every driver. The goal of SmPL on the other hand is to write a generic semantic patch that can transform all the relevant drivers, accommodating the variations among them. In this section and the following ones we describe the features of SmPL that make a semantic patch generic.

In the excerpt of the previous section, the reader may have wondered about the name `proc_info_func`, which indeed does not match the name of the `scsiglue` proc_info function, as shown in Figure 1, lines 3 and 4, or the names of any of the proc_info functions in the kernel source tree. Furthermore, the names of the parameters are not necessarily `buffer`, `start`, etc.; in particular, the introduced parameter `hostptr` is sometimes called simply `host`. To abstract away from these variations, SmPL provides *metavariables*. A metavariable is a variable that matches an arbitrary term in the driver source code. Metavariables are declared before the patch code specifying the transformation, between two `@@`s, borrowing the notation for delimiting line numbers in a traditional patch (Figure 1, lines 2 and 37). Metavariables are designated as matching terms of a specific kind, such as an `identifier`, `expression`,

or `statement`, or terms of a specific type, such as `int` or `off_t`. We call the combination of the declaration of a set of metavariables and a transformation specification a *rule*.

Back to our running example, the previous excerpt is made into a rule as follows:

```
@@
identifier proc_info_func;
identifier buffer, start, offset, inout, hostno;
identifier hostptr;
@@
  proc_info_func (
+       struct Scsi_Host *hostptr,
        char *buffer, char **start, off_t offset,
-       int hostno,
        int inout) { ... }
```

This code now amounts to a complete, valid semantic patch, although it still only performs part of our desired collateral evolution.

3.2.3 Multiple rules and inherited metavariables

The previous excerpt matches and transforms any function with parameters of the specified types. A proc_info function, however, is one that has these properties and interacts with the SCSI library in a specific way, namely by being provided by the driver to the SCSI library on the `proc_info` field of a `SHT` structure (for SCSI Host Template), which from the point of view of the SCSI library represents the device. To specify this constraint, we define another rule that identifies any assignment to such a field in the driver file. SmPL allows a semantic patch to define multiple rules, just as a traditional patch contains multiple regions separated by `@@`. The rules are applied in sequence, with each of them being applied to the entire source code of the driver. In our example we thus define one rule to identify the name of the callback function and another to transform its definition, as follows:

```
@ rule1 @
struct SHT ops;
identifier proc_info_func;
@@
  ops.proc_info = proc_info_func;

@ rule2 @
identifier rule1.proc_info_func;
identifier buffer, start, offset, inout, hostno;
identifier hostptr;
@@
  proc_info_func (
+       struct Scsi_Host *hostptr,
        char *buffer, char **start, off_t offset,
-       int hostno,
        int inout) { ... }
```

In the new semantic patch, the metavariable `proc_info_func` is defined in the first rule and referenced in the second rule, where we expect it to have the same value, which is enforced by `spatch`. In general, a rule may declare new metavariables and *inherit* metavariables from previous rules. Inheritance is explicit, in that the inherited metavariable must be declared again in the inheriting rule, and is associated with the name of the rule from which its value should be inherited (the rule name is only used in the metavariable declaration, but not in the transformation specification, which retains the form of ordinary C code). To allow this kind of inheritance, we must have means of naming rules. As shown in the semantic patch above, the name of a rule is placed between the two `@@`s at the beginning of a metavariable declaration. A name is optional, and is not needed if the rule does not export any metavariables.

Note that the first rule does not perform any transformations. Instead, its only role is to bind the `proc_info_func` metavariable to constrain the matching of the second rule. Once a metavariable obtains a value it keeps this value until the end of the current rule and in any subsequent rules that inherit it. Metavariables thus not only make a semantic patch generic by abstracting away from details of the driver code, but also allow communicating information and constraints from one part of the semantic patch to another, *e.g.*, from '–' code to '+' code, or from one rule to another.

A metavariable may take on multiple values, if the rule matches at multiple places in the driver code. If such a metavariable is inherited, the inheriting rule is applied once for each possible set of bindings of the metavariables it inherits. For example, in our case, a driver may set the `proc_info` field multiple times, to different functions, in which case rule 2 would be applied multiple times, for the names of each of them.

3.2.4 Sequences

So far, we have only considered the collateral evolutions on the header of the proc_info function. But collateral evolutions are needed in its body as well: deleting the

calls to scsi_host_hn_get and scsi_host_put, deleting the local variable holding the result of calling scsi_host_hn_get and the error checking code on its value. The affected code fragments are scattered throughout the body of the proc_info function and are separated from each other by arbitrary code specific to each SCSI driver. To abstract away from these irrelevant variations, SmPL provides the "..." operator, which matches any *sequence* of code. Refining rule2 of the semantic patch to perform these collateral evolutions gives:

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

The second rule of the semantic patch now has the form of the definition of a function that first contains a declaration of the hostptr variable, then a call to the function scsi_host_hn_get, then an error check, and finally a call to scsi_host_put sometime before the end. In practice, however, a proc_info function may *e.g.* contain many calls to scsi_host_put, as illustrated by the scsiglue example (Figure 1, lines 27 and 42). Closer inspection of the original scsiglue source code, however, shows that at execution time, the driver only executes one or the other of these calls to scsi_host_put, as the one on line 27 is only executed in an error case, and the one on line 42 is only executed in a non-error case. This is illustrated by Figure 2, which shows part of the control-flow graph of the scsiglue proc_info function. Because the execution pattern declare/scsi_host_hn_get/error-check/scsi_host_put is what must be followed by every SCSI proc_info driver, it is this pattern that the semantic patch should match against. The operator "..." thus matches paths in the control-flow graph rather than an arbitrary block of code in the driver source code. Thus, in practice, a single minus or plus line in the semantic patch can delete or add multiple lines in the source code of the driver.

The transformation specified in a rule is applied on driver code only if the whole rule matches code, not if only parts of the rule match code. Thus, here, the rule only matches proc_info callback functions having 5 parameters of the specified types, *and* the sequence of instructions declare/scsi_host_hn_get/error-check/scsi_host_put, *and* where these instructions share the use of the same variable, represented in the semantic patch by the repeated use of the same metavariable hostptr.

As said in the previous section, the repeated use of the same metavariable, here hostptr, can serve multiple purposes. First, it is used here to constrain some transformations by forcing two pieces of code to be equal in the driver code. So, for example, not all conditionals will be removed in the driver, only those testing the local structure returned by scsi_host_hn_get. Metavariables are also used to move code from one place to another. Here hostptr is used to move the matched local variable name into the parameter list. Metavariables declared as expression or statement can be used to move more complex terms.

3.2.5 Nested Sequences

The last transformation concerning the proc_info function is the replacement of every reference to the dropped hostno parameter by a field access. SmPL provides the <... ...> operator to perform such universal replacements. This operator is analogous to the /g operator of Perl. In order to avoid having to consider how references to hostno may interleave with the calls to scsi_host_hn_get and scsi_host_put, etc., we define a third rule that simply makes this transformation everywhere it applies:

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

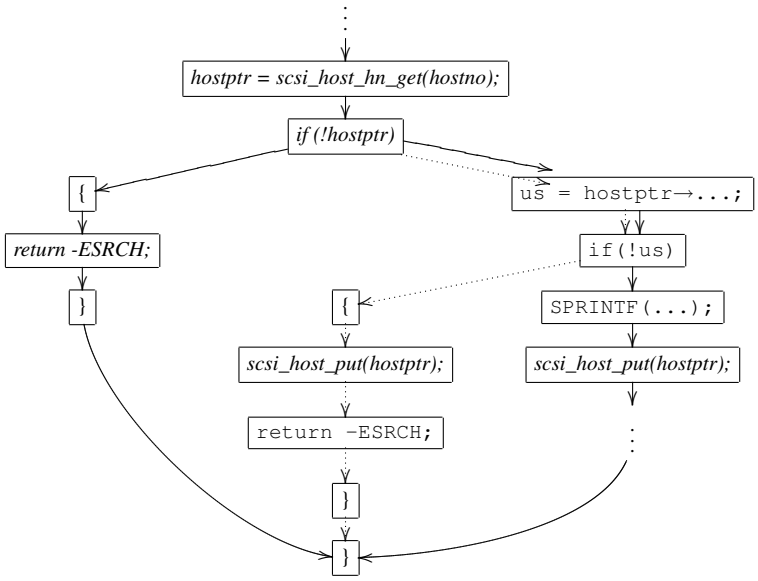

Figure 2: Simplified control-flow graph for part of Figure 1

Note that the operator "`...`" can be used to represent any kind of sequence. Here, in the function header, it is used to represent a sequence of parameters. It can also be used to provide flexible matching in initializers and structure definitions.

3.2.6 Isomorphisms

We have already mentioned that a semantic patch is insensitive to spacing, indentation and comments. Moreover, by defining sequences in terms of control-flow paths, we abstract away from the various ways of sequencing instructions that exist in C code. These features help make a semantic patch generic, allowing the patch developer to specify only a few scenarios, while `spatch` handles other scenarios that are semantically equivalent.

Other differences that we would like to abstract away from include variations within the use of specific C constructs. For example, if x is any expression that has pointer type, then $!x$, x `==` `NULL`, and `NULL` `==` x are all equivalent. For this, we provide a variant of the SmPL syntax for defining *isomorphisms*, sets of syntactically different terms that have the same semantics. The null pointer-test isomorphism is defined in this variant of SmPL as follows:

```
// iso file, not a semantic patch
@@ expression *X; @@
X == NULL <=> !X <=> NULL == X
```

Given this specification, the pattern `if(!hostptr)` in the semantic patch matches a conditional in the driver code that tests the value of `hostptr` using any of the listed variants.

In addition to a semantic patch, `spatch` accepts a file of isomorphisms as an extra argument. A file of isomorphisms is provided with the `spatch` distribution, which contains 30 equivalences commonly found in driver code. Finally, it is possible to specify that a single rule should use only the isomorphisms in a specific file, *file*, by annotating the rule name with `using` *file*.

3.3 All Together Now

The complete semantic patch for the proc_info collateral evolutions is shown below. As compared to the rules described above, this semantic patch contains an additional rule, `rule4`, which adjusts any calls to the proc_info function from within the driver. Note that in this rule, the metavariables that were declared as identifiers in rule2 to represent the parameters of the proc_info function are redeclared as `expression`s, to represent the proc_info function's arguments.

The second rule has also been slightly modified, in that two lines have been annotated with the "?" operator stating that those lines may or may not be present in the driver. Indeed, many drivers forget to check the return value of `scsi_host_hn_get` or forget to release

the structure before exiting the function. As previously noted, the latter omission is indeed what motivated the proc_info evolution.

Note that there is no rule for updating the prototype of the proc_info function, if one is contained in the file. When the type of a function changes, `spatch` automatically updates its prototype, if any.

```
@ rule1 @
struct SHT ops;
identifier proc_info_func;
@@
  ops.proc_info = proc_info_func;

@ rule2 @
identifier rule1.proc_info_func;
identifier buffer, start, offset, inout, hostno;
identifier hostptr;
@@
  proc_info_func (
+       struct Scsi_Host *hostptr,
        char *buffer, char **start, off_t offset,
-       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_func;
identifier rule2.hostno;
identifier rule2.hostptr;
@@
  proc_info_func(...) {
    <...
-   hostno
+   hostptr->host_no
    ...>
  }

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

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

On the 30 isomorphisms we have written, 3 of them "apply" to this semantic patch, accommodating many variations among the 19 drivers inside the kernel source tree. We have already mentioned the different ways to write a test such as `if(!hostptr)` in the previous section. There is also the various ways to assign a value in a field, which can be written `ops.proc_info = fn` as in our semantic patch, or written `ops->proc_info = fn` in some drivers, or written using a global structure initializer. Indeed, the last case was used for the `scsiglue.c` driver as shown by the following excerpt of this driver:

```
struct SHT usb_stor_host_template = {
  /* basic userland interface stuff */
  .name = "usb-storage",
  .proc_name = "usb-storage",
  .proc_info = usb_storage_proc_info,
  .proc_dir = NULL,
```

Finally, braces are not needed in C code when a branch contains only one statement. So, the pattern `{ ... return ...; }` in rule2 also matches a branch containing only the return statement.

It takes 23 seconds to `spatch` given the whole semantic patch to correctly update the 19 relevant drivers. If run on all the 2404 driver files inside the kernel source tree, it takes `spatch` 3 minutes to correctly update the same 19 drivers.

4 More Features, More Examples

So far we have written and tested 49 semantic patches for collateral evolutions found in the Linux 2.5 and Linux 2.6. By comparing the results produced by the semantic patch to the results produced by the traditional patch, we have found that `spatch` updates 92% of the driver files affected by these collateral evolutions correctly. In the remaining cases, there is typically a problem parsing the driver code, or needed information is missing because `spatch` currently does not parse header files. Parsing the driver code is a particular problem in our case, because our goal is to perform a source-to-source transformation, which means that we have chosen not to expand macros and preprocessor directives, and instead parse them directly.

In this section, we consider some other examples from our test suite, to illustrate some typical issues in semantic patch development.

4.1 Replacing one function name by another

In Linux 2.5.22, the function `end_request` was given a new first argument, of type `struct request *`. In practice, the value of this argument should be the next request from one of the driver's queue, as represented by a reference to the macro `CURRENT`. This collateral evolution affected 27 files spread across the directories `acorn`, `block`, `cdrom`, `ide`, `mtd`, `s390`, `sbus`.

The following semantic patch implements this collateral evolution:

```
@@ expression X; @@
- end_request(X)
+ end_request(CURRENT,X)
```

This semantic patch updates the 27 affected files in the Linux source tree correctly.

This example may seem almost too simple to be worth writing an explicit specification, as one can *e.g.* write a one-line `sed` command that has the same effect. Nevertheless, such solutions are error prone: we found that in the file `drivers/block/swim_iop.c`, the transformation was applied to the function `swimiop_send_request`, which has no relation to this collateral evolution. We conjecture that this is the result of applying a `sed` command, or some similar script, that replaces calls to `end_request` without checking whether this string is part of a more complicated function name. `spatch` enforces the syntactic structure of semantic patch code, allowing matches on identifier, expression, statement, etc. boundaries, rather than simply accepting anything that a superstring of the given pattern.

4.2 Collecting scattered information

In Linux 2.5.7, the function `video_generic_ioctl`, later renamed `video_usercopy`, was introduced to encapsulate the copying to and from user space required by `ioctl` functions. Ioctl functions allow the user level to configure and control a device, as they accept commands from the user level and perform the corresponding action at the kernel level. Without `video_usercopy`, an ioctl function has to use functions such as `copy_from_user` or `get_user` to access data passed in with the command, and functions such as `copy_to_user` or `put_user` to return information to the user level. With `video_usercopy`, the ioctl function receives a pointer to a kernel-level data structure containing the user-level arguments and can modify this data structure to return any values to user level.

Making an ioctl function `video_usercopy`-ready involves the following steps:

- Adding some new parameters to the function.
- Eliminating calls to `copy_from_user`, `put_user`, etc.
- Changing the references to the local structure used by these functions to use the pointer prepared by `video_usercopy`.

The last two points are somewhat complex, because the various commands interpreted by the ioctl function may each have their own requirements with respect to the user-level data. A command may or may not have a user-level argument, and it may or may not return a result to the user level. In the case where there is no use or returned value then no transformation should be performed; in the other cases, the structure containing the user-level argument or result should be converted to a pointer. Furthermore, there are multiple possible copying functions, and there are multiple forms that the references to the copied data can take.

Figure 3 shows a semantic patch implementing this transformation, under the simplifying assumption that the kernel-level representation of the user-level data is stored in a locally declared structure. This semantic patch consists of a single rule that changes the prototype of this function (adding some new variables, as indicated by `fresh identifier`), changes the types of the local structures, and removes the copy functions.

The many variations in an ioctl function noted above are visible in this rule. To express multiple possibilities, SmPL provides a *disjunction operator*, which begins with an open parenthesis in column 0, contains a list of possible patterns separated by a vertical bar in column 0, and then ends with a close parenthesis in column 0. Most of the body of the ioctl function pattern is represented as one large disjunction that considers the possibility of there being both a user-level argument and a user-level return value (lines 14-39), the possibility of there being a user-level argument but no user-level

```
1 @@
2 identifier ioctl, dev, cmd, arg, v, fld;
3 fresh identifier inode, file;
4 expression E, E1, e1,e2,e3;
5 type T;
6 @@
7   ioctl(
8 -       struct video_device *dev,
9 +       struct inode *inode, struct file *file,
10        unsigned int cmd, void *arg) {
11 +     struct video_device *dev = video_devdata(file);
12       ...
13 (
14 -     T v;
15 +     T *v = arg;
16       ...
17 (
18 -     if (copy_from_user(&v,arg,E)) { ... return ...; }
19 |
20 -     if (get_user(v,(T *)arg)) { ... return ...; }
21 )
22       <...
23 (
24 -     v.fld
25 +     v->fld
26 |
27 -     &v
28 +     v
29 |
30 -     v
31 +     *v
32 )
33       ...>
34 (
35 -   if (copy_to_user(arg,&v,E1)) { ... return ...; }
36 |
37 -   if (put_user(v,(T *)arg)) { ... return ...; }
38 )
39     ...
40 |
41 // a copy of the above pattern with the copy_to_user/put_user
42 // pattern dropped
43 |
44 // a copy of the above pattern with the copy_from_user/get_user
45 // pattern dropped
46 |
47   ... when != \(copy_from_user(e1,e2,e3)\|copy_to_user(e1,e2,e3)
48              \|get_user(e1,e2)\|put_user(e1,e2)\)
49 )
50   }
```

Figure 3: Semantic patch for the `video_ usercopy` collateral evolution

return value (elided in comments on line 41), the possibility of there being a user-level return value but no user-level argument (elided in comments on line 41-42), and there being neither a user-level argument nor a user-level return value (line 44-45). These possibilities are considered from top to bottom, with only the first one that matches being applied. This strategy is convenient in this case, because *e.g.* code using both a user-level argument and a user-level return value also matches all of the other patterns. The last cases uses "`...`" with the construct `when`. The `when` construct indicates a pattern that should not be matched anywhere in the code matched by the associated "`...`".

Within each of the branches of the outermost disjunction, there are several nested disjunctions. First, another disjunction is used to account for the two kinds of copy functions, `copy_from_user` or `get_user`. This case does not rely on the top-to-bottom strategy, because the patterns are disjoint. Next, between any copying, there is a nest (see Section 3.2.5) replacing the different variations on how to refer to a structure by the pointer-based counterpart. Here again, the ordering of the disjunction is essential, as the final case, `v`, should only be used when the variable is not used in a field access or address expression. Finally, there is a third disjunction allowing either `copy_to_user` or `put_user`.

Like the proc_info semantic patch, this semantic patch relies on isomorphisms. Specifically, the calls to the copy functions may appear alone in a conditional test as shown, or may be compared to 0, and as in the proc_info case, the return pattern in each of the conditional branches can match a single return statement, without braces.

4.3 Collecting scattered information

In Linux 2.6.20, the strategy for creating work queues and setting and invoking their callback functions changed as follows:

- Previously, all work queues were declared with some variant of `INIT_WORK`, and then could choose between delayed or undelayed work dynamically, by using either some variant of `schedule_work` or some variant of `schedule_delayed_work`. Since the changes in Linux 2.6.20, the choice between delayed or undelayed work has to be made statically, by creating the work queue with either `INIT_DELAYED_WORK` or `INIT_WORK`, respectively.
- Previously, creation of a work queue took as arguments a queue, a callback function, and a pointer to the value to be passed as an argument to the callback function. Since 2.6.20, the third argument is dropped, and the callback function is simply passed the work queue as an argument. From this, it can access the local data structure containing the queue, which can itself store whatever information was required by the callback function.

For simplicity, we consider only the case where the work queue is created using `INIT_WORK`, where it is the field of a local structure, and where the callback function

passed to `INIT_WORK` expects this local structure as an argument.

Figure 4 shows the semantic patch. In this semantic patch, we name all of the rules, to ease the presentation, but only those with descriptive names, such as `is_delayed`, are necessary.

The semantic patch is divided into two sections, the first for the case where the work queue is somewhere used with a delaying function such as `schedule_delayed_work` and the second for the case where such a function is not used on the work queue. Both cases can occur within a single driver, for different work queues. The choice between these two variants is made at the first rule, `is_delayed`, using a trick based on metavariable binding. This rule matches all calls to `schedule_delayed_work` and other functions indicating delayed work, for any work queue `&device->fld` and arbitrary task `E`. The next five rules, up to the commented dividing line, refer directly or indirectly to the type of the structure containing the matched work queue `&device->fld`, and thus these rules are only applied to work queues for which the match in `is_delayed` somewhere succeeds. The remaining three rules, at the bottom of the semantic patch, do not depend on the rule `is_delayed`, and thus apply to work queues for which there is no call to any delaying function.

In the first half of the semantic patch, the next task is to convert any call to a non-delaying work queue function to a delaying one, by adding a delay of 0 (`rule2`). The rule `delayed_fn` then changes calls to `INIT_WORK` to calls to `INIT_DELAYED_WORK` and adjusts the argument lists such that the cast on the second argument (the work queue callback function) is dropped and the third argument is dropped completely. Note that the casts on the second and third arguments need not be present in the driver code, thanks to an isomorphism. Next (`rule4`), the work queue is changed from having type `work_struct` to having type `delayed_work`. The last two rules of this section, `rule5` and `rule5a`, update the callback functions identified in the call to `INIT_WORK`. The first, `rule5`, is for the case where the current parameter type is `void *` and the second, `rule5a`, is for the case where the parameter type is the type of the local structure containing the work queue. In both cases, the parameter is given the type `struct work_struct`, and then code using the macro `container_of` is added to the body of the function to reconstruct the original argument value.

```
@ is_delayed @
type local_type; local_type *device; expression E,E1;
identifier fld;
@@
( schedule_delayed_work(&device->fld,E)
| cancel_delayed_work(&device->fld)
| schedule_delayed_work_on(E1,&device->fld,E)
| queue_delayed_work(E1,&device->fld,E)
)

@ rule2 @
is_delayed.local_type *device;
identifier is_delayed.fld; expression E1;
@@
(
- schedule_work(&device->fld)
+ schedule_delayed_work(&device->fld,0)
|
- schedule_work_on(E1,&device->fld)
+ schedule_delayed_work_on(E1,&device->fld,0)
|
- queue_work(E1,&device->fld)
+ queue_delayed_work(E1,&device->fld,0)
)

@ delayed_fn @
type T,T1; identifier is_delayed.fld, fn;
is_delayed.local_type *device;
@@
- INIT_WORK(&device->fld, (T)fn, (T1)device);
+ INIT_DELAYED_WORK(&device->fld, fn);

@ rule4 @
type is_delayed.local_type; identifier is_delayed.fld;
@@
local_type { ...
-          struct work_struct fld;
+          struct delayed_work fld;
           ... };

@ rule5 @
identifier data, delayed_fn.fn, is_delayed.fld;
type T, is_delayed.local_type; fresh identifier work;
@@
- fn(void *data) {
+ fn(struct work_struct *work) {
    <...
-   (T)data
+   container_of(work,local_type,fld.work)
    ...>
  }

@ rule5a @
identifier data, delayed_fn.fn, is_delayed.fld;
type is_delayed.local_type; fresh identifier work;
@@
- fn(local_type *data) {
+ fn(struct work_struct *work) {
+   local_type *data = container_of(work,local_type,fld.work);
    ...
  }
//----------------------------------------------------------
@ non_delayed_fn @
type local_type, T,T1; local_type *device; identifier fld, fn;
@@
- INIT_WORK(&device->fld, (T)fn, (T1)device);
+ INIT_WORK(&device->fld, fn);

@ rule7 @
identifier data, non_delayed_fn.fn, non_delayed_fn.fld;
type T, non_delayed_fn.local_type; fresh identifier work;
@@
- fn(void *data) {
+ fn(struct work_struct *work) {
    <...
-   (T)data
+   container_of(work,local_type,fld)
    ...>
  }

@ rule7a @
identifier data, non_delayed_fn.fn, non_delayed_fn.fld;
type non_delayed_fn.local_type; fresh identifier work;
@@
- fn(local_type *data) {
+ fn(struct work_struct *work) {
+   local_type *data = container_of(work,local_type,fld);
    ...
  }
```

Figure 4: Semantic patch for the `INIT_WORK` collateral evolution

In the second section, calls to `INIT_WORK` for non-delayed work queues have their second and third arguments transformed as in the delayed case. The rules `rule7` and `rule7a` then update the callback functions analogously to rules `rule5` and `rule5a`.

Using the Linux 2.6 git repository [3], we have identified 245 driver files that use work queues. Of these, 45% satisfy the assumptions on which this semantic patch is based. This semantic patch applies correctly to 91% of them. The remaining cases are due to some constructs that are not treated adequately by our approach, to inter-file effects, and to some optimizations made by the programmer that are too special-purpose to be reasonable to add to a generic transformation rule.

5 Conclusion

In this paper we have presented SmPL, our scripting language to automate and document collateral evolutions in Linux device drivers. This language is based on the patch syntax, familiar to Linux developers, but accommodates many variations among drivers. As opposed to a traditional patch, a single semantic patch can update hundreds of drivers at thousands of code sites because of the features of SmPL, including the use of metavariables, isomorphisms, and control-flow paths, which makes a semantic patch generic. We hope that the use of semantic patches will make collateral evolutions in Linux less tedious and more reliable. We also hope that it will help developers with drivers outside the kernel source tree to better cope with the fast evolution of Linux.

Until now we have tried to replay what was already done by Linux programmers. We would like now to interact with the Linux community and really contribute to Linux by implementing or assisting library developers in performing new evolutions and collateral evolutions. As a first step we have subscribed to the janitors kernel mailing list and planned to contribute by automating some known janitorings [4]. We would also like to investigate if SmPL could be used to perform collateral evolutions in other Linux subsystems such as filesystems or network protocols or to perform other kinds of program transformations.

Finally, introducing semantic patches in the development process may lead to new processes, or new tools. For instance, how can semantic patches be integrated in versioning tools such as `git`. We could imagine a versioning tool aware of semantic patches and of the semantics of C, that could for example automatically update new drivers coming from outside the kernel source tree with respect to some recent semantic patches. Semantic patches, due to their degree of genericity, can also help with the problem of conflicts between multiple patches that are developed concurrently and affect some common lines of code, but in an orthogonal way. Finally, for the same reason, semantic patches should be more portable from one Linux version to the next, in the case of a patch that is not immediately accepted into the Linux kernel source tree.

All the semantic patches we have written, as well as a binary version of `spatch`, are available on our website: `http://www.emn.fr/x-info/coccinelle`. Reading those semantic patches can give a better feeling of the expressivity of SmPL. They can also be used as a complement to this tutorial.

References

[1] "Understanding Collateral Evolution in Linux Device Drivers." Yoann Padioleau, Julia L. Lawall, and Gilles Muller. *Proceedings of the ACM SIGOPS EuroSys 2006 Conference*, Leuven, Belgium, April, 2006, pages 59–71.

[2] `http://lwn.net/Articles/36311/`.

[3] `http://git.kernel.org/git/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary`.

[4] `http://kernelnewbies.org/KernelJanitors/Todo`.

Proceedings of the Linux Symposium

Volume Two

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*