SmPL: A Domain-Specific Language for Specifying Collateral Evolutions in Linux Device Drivers

Yoann Padioleau [a], Julia L. Lawall [b] and Gilles Muller [a]

[a] *OBASCO Group, Ecole des Mines de Nantes-INRIA, LINA, Nantes, France*
{Yoann.Padioleau,Gilles.Muller}@emn.fr

[b] *DIKU, University of Copenhagen, Copenhagen, Denmark*
julia@diku.dk

Abstract

Collateral evolutions are a pervasive problem in large-scale software development. Such evolutions occur when an evolution that affects the interface of a generic library entails modifications, *i.e.*, collateral evolutions, in all library clients. Performing these collateral evolutions requires identifying the affected files and modifying all of the code fragments in these files that in some way depend on the changed interface.

We have studied the collateral evolution problem in the context of Linux device drivers. Currently, collateral evolutions in Linux are mostly done manually using a text editor, possibly with the help of tools such as `grep`. The large number of Linux drivers, however, implies that this approach is time-consuming and unreliable, leading to subtle errors when modifications are not done consistently.

In this paper, we propose a transformation language, SmPL, to specify collateral evolutions. Because Linux programmers are accustomed to exchanging, reading, and manipulating program modifications in terms of patches, we build our language around the idea and syntax of a patch, extending patches to *semantic patches*.

Key words: Linux, device drivers, collateral evolutions, domain-specific languages.

1 Introduction

One major difficulty, and the source of highest cost, in software development is to manage evolution. Software evolves to add new features, to adapt to new requirements, and to improve performance, safety, or the software architecture. Nevertheless, while evolution can provide long-term benefits, it can also introduce short-term difficulties, when the evolution of one component affects interfaces on which other components rely.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In previous work [16], we have identified the phenomenon of *collateral evolution*, in which an evolution that affects the interface of a generic library entails modifications, *i.e.*, collateral evolutions, in all library clients. As part of this previous work, we have furthermore studied this phenomenon in the context of Linux device drivers. Collateral evolutions are a significant problem in this context because device drivers make up over half of the Linux source code and are highly dependent on the kernel and various driver support libraries for functions and data structures. This previous study concluded by identifying a taxonomy of the kinds of collateral evolutions that are required in device drivers. These include changes in calls to driver support library functions to add or drop new arguments, changes in callback functions defined by drivers to add or drop required parameters, changes in data structures to add or drop fields, and changes in function usage protocols.

Performing collateral evolutions in Linux device drivers requires identifying the affected driver files and modifying all of the code fragments in these files that somehow depend on the changes in the driver support library interface. Standard techniques include manual search and replace in a text editor, tools such as `grep` to find driver files with relevant properties, and tools such as `sed`, `perl` scripts, and `emacs` macros to update affected driver code fragments. None of these approaches, however, provides any support for taking into account the syntax and semantics of C code. Errors result, such as deleting more lines of code than intended or overlooking some relevant code fragments. Furthermore, many collateral evolutions involve control-flow properties, and thus require substantial programming-language expertise to implement correctly.

In this paper, we propose a declarative transformation language, SmPL (Semantic Patch Language), to express precisely and concisely collateral evolutions of Linux device drivers. Linux programmers are accustomed to exchanging, reading, and manipulating patch files that provide a record of previously performed changes. Thus, we base the syntax of SmPL on the patch file notation. Unlike traditional patches, which record changes at specific sites in specific files, SmPL can describe generic transformations that apply to multiple collateral evolution sites. In particular, transformations are defined in terms of control-flow graphs rather than abstract syntax trees, and thus follow not the syntax of the C code but its semantics. We thus refer to the transformation rules expressed using SmPL as *semantic patches*.

SmPL is a first step in a larger project to develop a transformation tool, Coccinelle, providing automated assistance for performing collateral evolutions. This assistance will comprise the SmPL language for specifying collateral evolutions and a transformation engine for applying them to device driver code. We expect that when the developer of a driver support library modifies the library's interface, he will create the corresponding semantic patch, relying on his understanding of the protocol for using the affected interface elements and the structure of typical driver code. He will then distribute the semantic patch to driver maintainers who will use it to update their drivers. Our

goal is that the transformation process should be robust, and interactive when necessary. In particular, it should remain able to assist the driver maintainer even when an exact match of the rule against the source code is not possible, in the case of unexpected variations in driver coding style.

The rest of this paper is organized as follows. Section 2 describes a set of collateral evolutions that will be used as our running example. Section 3 illustrates how one of these collateral evolutions is expressed using the standard patch notation. Section 4 presents SmPL in terms of this example. Finally, Sections 5 and 6 present related work and conclusions, respectively.

2 Motivating Example

As a running example, we consider the collateral evolutions that took place in SCSI drivers in Linux 2.5.71, in each driver's "proc_info" function. Such a function is exported by a SCSI driver to the SCSI driver support library via the `proc_info` field of a `SHT` (for SCSI Host Template) structure. Each function prints information about the status of the corresponding device in a format compatible with the Linux `procfs` file system.

The collateral evolutions in the proc_info functions were triggered by the decision that it is undesirable for drivers to directly use the functions `scsi_host_hn_get` to obtain access to a representation of the device and `scsi_host_put` to give up this access, because any incorrect use of these functions can break the integrity of associated reference counts [11]. Starting in Linux 2.5.71, these functions were no longer exported by the SCSI driver support library. To compensate for this evolution, the proc_info functions were then passed a representation of the device as an extra argument. An existing parameter that was used as the argument of `scsi_host_hn_get` was also removed. Among the drivers in the Linux source tree, these collateral evolutions affect 19 SCSI driver files, in 4 different directories.

The collateral evolution in the case of the `scsiglue` driver is illustrated in Figure 1. As shown in Figure 1a, in Linux 2.5.70 the function `usb_storage_proc_info` declares a local variable `hostptr` (line 7), representing the device, and contains code to access (line 15), test (lines 16-18), and release (lines 23 and 33) the device value. All of this code is removed in Linux 2.5.71 (Figure 1b).[1] Instead, the local variable `hostptr` becomes a parameter of `usb_storage_proc_info`, with the same type. Additionally, the `hostno` parameter of `usb_storage_proc_info` in Linux 2.5.70 is dropped in Linux 2.5.71. References to `hostno` are replaced by accesses to the `host_no` field of the new `hostptr` parameter.

This example illustrates the combination of two of the basic kinds of collateral evolutions identified in our previous work [16]: (i) the introduction of

[1] The conditional on lines 21-25 is removed as well in Linux 2.5.71, but that appears to be related to another evolution, and thus we have left it in for the purposes of the example.

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

(a) Linux 2.5.70

```
1 static int usb_storage_proc_info (struct Scsi_Host *hostptr,
2          char *buffer, char **start, off_t offset,
3          int length, int inout)
4 {
5    struct us_data *us;
6    char *pos = buffer;
7
8    unsigned long f;
9
10   /* if someone is sending us data, just throw it away */
11   if (inout)
12      return length;
13
14
15
16
17
18
19   us = (struct us_data*)hostptr->hostdata[0];
20
21   /* if we couldn't find it, we return an error */
22   if (!us) {
23
24     return -ESRCH;
25   }
26
27   /* print the controller name */
28   SPRINTF("   Host scsi%d: usb-storage\n", hostptr->host_no);
29   /* print product, vendor, and serial number strings */
30   SPRINTF("       Vendor: %s\n", us->vendor);
31   ... // some code omitted
32
33
34   ... // some code omitted
35   return length;
36 }
```

(b) Linux 2.5.71

Fig. 1. An example of collateral evolution, in `drivers/usb/storage/scsiglue.c`

a new parameter and the corresponding elimination of computation that this parameter makes redundant, and (ii) the elimination of a parameter and the introduction of computations to reconstruct its value.

3 The Patch Approach

Traditionally, changes in the Linux operating system are published using patch files [12]. A patch file is created by manually performing the change in the source code, and then running the `diff` tool on the old and new versions, with special arguments so that `diff` records not only the differences, but also some position and context information. An entry in a patch file begins with a header, indicating the name of the old file preceded by `---` and the name of the new file preceded by `+++`. The header is followed by a sequence of regions, each beginning with `@@ ... @@`, which specifies the starting line numbers in the old and new files. A region then contains a sequence of lines of text, in which lines that are added are indicated by `+` in the first column, lines that are removed are indicated by `-` in the first column, and lines that provide context information are indicated by a space in the first column. To apply a patch file, each mentioned file is visited, and the indicated lines are added and removed.

Normally, a patch file is applied to a file that is identical to the one used by the Linux developer to create it. It is possible to instruct the `patch` tool to ignore the line numbers or some of the lines of context, to be able to apply a patch to a file that is similar but not identical to the one intended. Nevertheless, because there is no semantic analysis of either the meaning of

```
--- a/drivers/usb/storage/scsiglue.c Sat Jun 14 12:18:55 2003
+++ b/drivers/usb/storage/scsiglue.c Sat Jun 14 12:18:55 2003
@@ -264,33 +300,21 @@
-static int usb_storage_proc_info (
+static int usb_storage_proc_info (struct Scsi_Host *hostptr,
               char *buffer, char **start, off_t offset,
-              int length, int hostno, int inout)
+              int length, int inout)
 {
   struct us_data *us;
   char *pos = buffer;
-  struct Scsi_Host *hostptr;
   unsigned long f;

   /* if someone is sending us data, just throw it away */
   if (inout)
           return length;

-  /* find our data from the given hostno */
-  hostptr = scsi_host_hn_get(hostno);
-  if (!hostptr) {
-          return -ESRCH;
-  }
   us = (struct us_data*)hostptr->hostdata[0];

   /* if we couldn't find it, we return an error */
   if (!us) {
-          scsi_host_put(hostptr);
           return -ESRCH;
   }

   /* print the controller name */
-  SPRINTF("   Host scsi%d: usb-storage\n", hostno);
+  SPRINTF("   Host scsi%d: usb-storage\n", hostptr->host_no);
   /* print product, vendor, and serial number strings */
   SPRINTF("       Vendor: %s\n", us->vendor);

@@ -318,9 +342,6 @@
     *(pos++) = '\n';
  }

-  /* release the reference count on this host */
-  scsi_host_put(hostptr);

   /*
    * Calculate start of next buffer, and return value.
```

Fig. 2. Excerpt of the patch file from Linux 2.5.70 to Linux 2.5.71

the patch or that of the affected source code, this approach is error prone. Furthermore, in practice, patches are quite brittle, and variations in the source code imply that parts of the patch cannot be applied at all.

Figure 2 shows part of the patch file used to update the function `usb_storage_proc_info` from Linux 2.5.70 to Linux 2.5.71. While this patch may apply to minor variations of the `scsiglue.c` file, it cannot be applied to proc_info functions in other SCSI drivers, because of the `scsiglue`-specific names such as `usb_storage_proc_info` used in the modified lines of code. This is unfortunate, because multiple files have to be updated in the same way.

4 Expressing Collateral Evolutions as a Semantic Patch

To express collateral evolutions, we propose a new language SmPL as a means of generalizing patches to *semantic patches*. A semantic patch is a specification that visually resembles a patch, but whose application is based on the semantics of the code to be transformed, rather than its syntax. The complete language is defined in the appendix. Here, we present SmPL via an example, a semantic patch expressing the collateral evolutions described in Section 2. We develop the semantic patch incrementally, by showing successive excerpts that each illustrate a feature of SmPL. In contrast to a patch that applies to only one file, the semantic patch can be applied to all of the files in the Linux source tree, to selected files, to an individual file, or even to files outside the Linux source tree.

4.1 Replacement

Our first task is to change the function signature, to add an argument that points to a `Scsi_host` structure and to drop the `hostno` argument. We express these modifications in SmPL as follows:

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

As in a standard patch, the lines beginning with `+` and `-` are added and deleted, respectively. The remaining lines describe the modification context. This excerpt is applied throughout a file, and transforms every matching code fragment, regardless of the fragment's spacing, indentation or comments.

4.2 Metavariables, part 1

The previous rule assumes that the proc_info function has parameters `buffer`, `start`, etc. In practice, however, the parameter names vary from one driver to another. To make the rule insensitive to the choice of names, we replace the explicit names by *metavariables*. These are declared in a section delimited by `@@` that appears before each transformation, as illustrated below:

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

The metavariables `buffer`, `start`, `offset`, `length`, `hostno`, and `inout` are used on lines annotated with `-` or space, and thus match terms in the original source program. They are declared as `identifier`, indicating that they match any identifier. The metavariable `hostptr` represents a parameter that is newly added to the function signature. We thus declare it as a `fresh identifier`, indicating that some identifier should be chosen that does not conflict with the other identifiers in the program.

A semantic patch may contain multiple regions, each declaring some metavariables and specifying a transformation rule. Once declared, a metavariable obtains its value from matching the given transformation rule against the source code. It then keeps its value over subsequent regions until it is redeclared.

4.3 Metavariables, part 2

As illustrated in Figure 1, the name of the function to transform is generally not `proc_info_func`, but is something specific to each driver. Rather than rely on properties of the name chosen, we identify the function in terms of its relation with the SCSI interface. Specifically, the function to modify is the one that is stored in the `proc_info` field of a `SHT` structure. The following excerpt, placed before the excerpt of Section 4.2, expresses this constraint:

```
@@
struct SHT sht;
local function proc_info_func;
@@
   sht.proc_info = proc_info_func;
```

The declaration `struct SHT sht;` indicates that the metavariable `sht` represents an expression of type `struct SHT`. This type specification avoids ambiguity in the reference to the `proc_info` field when multiple structure types have fields of the same name. If there is more than one assignment of the `proc_info` field, the metavariable `proc_info_func` is bound to the set of all possible right-hand sides. Subsequent transformations that use this metavariable are instantiated for all elements of this set.

The rule above is written as a direct assignment of the `proc_info` field to the name of a local function. In the code to be transformed, however, the right-hand-side of this assignment could be some other expression that is the alias of a local function. The patterns of such aliasing that we have observed in driver code are very simple, such as initializing a local variable to a different function in each branch of a conditional, and then using this local variable immediately thereafter. Such aliases can be detected by a standard dataflow analysis.

4.4 Sequences, part 1

The next step is to remove the sequence of statements that declares the `hostptr` local variable and accesses, tests, and releases its value. In practice these statements can be separated by arbitrary code, as illustrated in Figure 1a (lines 7, 15-18, 23, and 33). To specify arbitrary sequences SmPL provides the operator "...", which we use as follows:

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

If we compare this rule to Figure 1a, we see that the declaration, access, and test of `hostptr` each appear exactly once in the source program, as in the rule, but that `scsi_host_put` is called twice, once in line 23 just before returning an error code, and once in line 33 near the end of the function. To address this case, sequences in SmPL describe sequences in the control-flow graph rather than sequences in the abstract-syntax tree. Specifically, when a transformation includes the operator "...", it is applied to every control flow path between the terms matching the endpoints, which here are the beginning and end of the function definition. For instance, in Figure 1a, after the assignment of the variable `us`, there are two control flow paths, one that is an error path (lines 23-24), and another that continues until the final return (lines 27-35). A call to `scsi_host_put` is removed from each of them. Thus, in practice, a single `-` line may erase multiple lines of code, one per control flow path.

Recall that in Section 4.2, we created a fresh identifier as the new parameter `hostptr`. In fact, when the collateral evolutions were performed by hand, the parameter was always given the name of the deleted `Scsi_Host`-typed local variable. Now that we have expanded the semantic patch excerpt to contain both the parameter and the local variable declaration, we can express this naming strategy by using the same metavariable, declared as an `identifier`, in both cases. This repetition implies that both occurrences refer to the same term, thus transmitting the name of the old local variable to the new parame-

ter. Metavariables are thus similar to logic variables, in that every occurrence of a metavariable within a rule refers to the same set of terms. Unlike the logic variables of Prolog, however, metavariables are always bound to ground terms.

The collateral evolution described in this section introduced some bugs in the Linux 2.5.71 version. For example, in two files the `hostno` parameter was not dropped, resulting in a function that expected too many arguments. This problem was fixed in Linux 2.6.0, which was released 6 months later.

4.5 Sequences, part 2

Finally, we consider the treatment of references to the deleted `hostno` parameter. In each case, the reference should be replaced by `hostptr->host_no`. Here we are not interested in enforcing any particular number of occurrences of `hostno` along any given control-flow path, so we use the operator `<... ...>` that applies the transformation everywhere within the matched region:

```
@@
@@
  proc_info_func(...) {
    <...
-   hostno
+   hostptr->host_no
    ...>
  }
```

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.

4.6 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 and taking into account data flow, we abstract away from the various ways of constructing *e.g.* loops and complex expressions that exist in C code. These features help make a semantic patch generic, allowing the patch developer to write only a few scenarios, while the transformation tool handles other scenarios that are semantically equivalent.

In fact, these features are a part of a larger set of semantic equivalences that we refer to as *isomorphisms*. Other isomorphisms that are relevant to this example include typedef aliasing (*e.g.*, `struct SHT` is commonly referred to as `SCSI_Host_Template`), the various ways of referencing a structure field (*e.g.*, *exp*`->`*field* and `*`*exp*`.`*field*), and the various ways of testing for a null pointer (*e.g.*, `!hostno` and `hostno == NULL`). We have identified many more useful isomorphisms, and continue to discover new ones.

4.7 All together now

Figure 3 presents the complete semantic patch that implements the collateral evolutions described in Section 2. This version is augmented as compared to the previous excerpts in that the error checking code `if (!hostptr) { ... }` and the call to `scsi_host_put` are annotated with `?`, indicating that matching these patterns is optional (although removing them if they are matched is obligatory). The `?` annotation is often useful with error checking code, as studies such as that of Engler et al. [1] show that error checking code is often (incorrectly) omitted in device drivers.

```
@@
struct SHT sht;
local function proc_info_func;
@@
   sht.proc_info = proc_info_func;

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

@@
@@
  proc_info_func(...) {
    <...
-   hostno
+   hostptr->host_no
    ...>
 }
```

Fig. 3. A complete Semantic Patch

4.8 Assessment

Considering Figure 3, it is apparent that much of the description of the collateral evolution is in terms of ordinary C code. Among the 62 semantic patches we have written, we have often found it possible to construct a semantic patch by copying and modifying existing driver code. The close relationship to actual driver code should furthermore make it easy for a driver maintainer who

wants to apply a semantic patch to understand its intent and the relationship between the various transformed terms.

The "proc_info" semantic patch applies to 19 files in 4 different directories of the Linux source tree. In the standard patch notation, the specification of the required changes amounts to 614 lines of code for the files in the Linux source tree, resulting in on average 32.3 lines per file. The semantic patch is 33 lines of code and applies to all relevant files including those not in the Linux source tree. Because semantic patches are intended to implement collateral evolutions, which are determined by interface changes, and because interface elements are typically used only according to very restricted protocols, we expect that most semantic patches will exhibit a similar degree of reusability. Indeed, in our previous study of collateral evolutions in over 1600 driver files [16], we have found that there is little variation in the structure of the code affected by a given evolution, an observation that is further substantiated by another study of driver code [10].

5 Related work

Influences. The design of SmPL was influenced by a number of sources. Foremost among these is our target domain, the world of Linux device drivers. Linux programmers manipulate patches extensively, have designed various tools around them [13], and use its syntax informally in e-mail to describe software evolutions. This has encouraged us to consider the patch syntax as a valid alternative to classical rewriting systems. Other influences include the *Structured Search and Replace* (SSR) facility of the IDEA development environment from JetBrains [14], which allows specifying patterns using metavariables and provides some isomorphisms, and the work of De Volder on JQuery [3], which uses Prolog logic variables in a system for browsing source code. Finally, we were inspired to base the semantics of SmPL on control-flow graphs rather than abstract syntax trees by the work of Lacey and de Moor on formally specifying compiler optimizations. [8]

Other work. Refactoring is a generic program transformation that reorganizes the structure of a program without changing its semantics [6]. Some of the collateral evolutions in Linux drivers can be seen as refactorings. Refactorings, however, apply to the whole program, requiring accesses to all usage sites of affected definitions. In the case of Linux, however, the entire code base is not available, as many drivers are developed outside the Linux source tree. There is currently no way of expressing or generating the effect of a refactoring on such external code. Other collateral evolutions are very specific to an OS API, and thus cannot be described as part of a generic refactoring [9]. Moreover, in practice, refactorings are used via a development environment such as Eclipse that only provides a fixed set of transformations. JunGL is a scripting language that allows programmers to implement new refactorings [20]. This lan-

guage should be able to express collateral evolutions. Nevertheless, a JunGL transformation rule does not follow the structure of the source terms, and thus does not make visually apparent the relationship between transformed terms, which we have found makes the provided examples difficult to read. Furthermore, the language is in the spirit of ML, which is not part of the standard toolbox of Linux developers.

A number of program transformation frameworks have recently been proposed, targeting industrial-strength languages such as C and Java. CIL [15] and XTC [7] are essentially parsers that provide some support for implementing abstract syntax tree traversals. No program transformation abstractions, such as pattern matching using logic variables, are currently provided. CIL also manages the C source code in terms of a simpler intermediate representation. Rewrite rules must be expressed in terms of this representation rather than in terms of the code found in a relevant driver. Stratego is a domain-specific language for writing program transformations [21]. Convenient pattern-matching and rule management strategies are built in, implying that the programmer can specify what transformations should occur without cluttering the code with the implementation of transformation mechanisms. Nevertheless, only a few program analyses are provided. Any other analyses that are required, such as control-flow analysis, have to be implemented in the Stratego language. In our experience, this leads to rules that are very complex for expressing even simple collateral evolutions.

Coady et al. have used Aspect-Oriented Programming (AOP) to extend OS code with new features [2,5]. Nevertheless, AOP is targeted towards modularizing concerns rather than integrating them into a monolithic source code. In the case of collateral evolutions, our observations suggest that Linux developers favor approaches that update the source code, resulting in uniformity among driver implementations. For example, on occasion, wrapper functions have been introduced to allow code respecting both old and new versions of an interface to coexist, but these wrapper functions have typically been removed after a few versions, when a concerted effort has been made to update the code to respect the new version of the interface.

The Linux community has recently begun using various tools to better analyze C code. Sparse [18] is a library that, like a compiler front end, provides convenient access to the abstract syntax tree and typing information of a C program. This library has been used to implement some static analyses targeting bug detection, building on annotations added to variable declarations, in the spirit of the familiar `static` and `const`. Smatch [19] is a similar project and enables a programmer to write Perl scripts to analyze C code. Both projects were inspired by the work of Engler et al. [4] on automated bug finding in operating systems code. These examples show that the Linux community is open to the use of automated tools to improve code quality, particularly when these tools build on the traditional areas of expertise of Linux developers.

6 Conclusion

In this paper, we have proposed a declarative language, SmPL, for expressing the transformations required in performing collateral evolutions in Linux device drivers. This language is based on the patch syntax familiar to Linux developers, but enables transformations to be expressed in a more general form. The use of isomorphisms in particular allows a concise representation of a transformation that can nevertheless accommodate multiple programming styles. SmPL furthermore addresses all of the elements of the taxonomy of the kinds of collateral evolutions in Linux device drivers identified in our previous work.

We are currently completing a formal specification of the semantics of SmPL, and are exploring avenues for an efficient implementation. In the longer term, we plan to use SmPL to specify the complete set of collateral evolutions required to update drivers from one version of Linux to a subsequent one.

Acknowledgments
This work has been supported in part by the Agence Nationale de la Recherche (France) and the Danish Research Council for Technology and Production Sciences. Further information about the Coccinelle project can be found at the URL: `http://www.emn.fr/x-info/coccinelle/`

References

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001.

[2] Y. Coady and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003*, pages 50–59, Boston, Massachusetts, Mar. 2003.

[3] K. De Volder. JQuery: A generic code browser with a declarative configuration language. In *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006*, pages 88–102, Charleston, SC, Jan. 2006.

[4] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, Oct. 2000.

[5] M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. Patch (1) considered harmful. In *10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, June 2005.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[7] R. Grimm. XTC: Making C safely extensible. In *Workshop on Domain-Specific Languages for Numerical Optimization*, Argonne National Laboratory, Aug. 2004.

[8] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001*, number 2027 in Lecture Notes in Computer Science, pages 52–68, Genova, Italy, Apr. 2001.

[9] J. L. Lawall, G. Muller, and R. Urunuela. Tarantula: Killing driver bugs before they hatch. In *The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 13–18, Chicago, IL, Mar. 2005.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–302, San Fransisco, CA, Dec. 2004.

[11] LWN. ChangeLog for Linux 2.5.71, 2003. http://lwn.net/Articles/36311/.

[12] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003. Unified Format section, http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html.

[13] A. Morton. Patch management scripts, Oct. 2002. Available at http://www.zip.com.au/~akpm/linux/patches/.

[14] M. Mossienko. Structural search and replace: What, why, and how-to. *OnBoard Magazine*, 2004. http://www.onboard.jetbrains.com/is1/articles/04/10/ssr/.

[15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction, 11th International Conference, CC 2002*, number 2304 in Lecture Notes in Computer Science, pages 213–228, Grenoble, France, Apr. 2002.

[16] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 59–71, Leuven, Belgium, Apr. 2006.

[17] F. Pottier and Y. Régis-Gianas. Menhir reference manual. version 20060505. Available at http://cristal.inria.fr/ fpottier/menhir/manual.pdf.

[18] D. Searls. Sparse, Linus & the Lunatics, Nov. 2004. Available at http://www.linuxjournal.com/article/7272.

[19] The Kernel Janitors. Smatch, the source matcher, June 2002. Available at http://smatch.sourceforge.net.

[20] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[21] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. Spinger-Verlag, 2004.

A The SmPL Grammar

This section presents the SmPL grammar. This definition follows closely our implementation using the Menhir parser generator [17].

The grammar uses some rules where the left-hand side is in all capital letters. These are macros, which take one or more grammar rule right-hand-sides as arguments. The grammar also uses some unspecified nonterminals, such as id, const, etc. These refer to the sets suggested by the name, *i.e.*, id refers to the set of possible C-language identifiers, while const refers to the set of possible C-language constants.

Program

program ::= (*metavariables* [--- filename +++ filename] *transformation*)$^+$

Between the metavariables and the transformation rule, there can be a specification of constraints on the names of the old and new files, analogous to the filename specifications in the standard patch syntax (see Figure 2).

Metavariables

Fresh metavariables must only be used in + code. Metavariables must occur at least once in the transformation immediately following their declaration. These properties are not expressed in the grammar, but are checked by a subsequent analysis.

metavariables ::= `@@` *metadec** `@@`

metadec ::= [`!` | `?` | `+`] *metakind* (id `,`)* id `;`

metakind ::= [`fresh`] `identifier` | `type` | `parameter` [`list`] | `error`
| `expression` [`list`] | `statement` [`list`] | [`local`] `function`
| [`constant`] *metaexptype* | `constant`

metaexptype ::= *type* | `{` (*type* `,`)* *type* `}`

Subsequently, we refer to arbitrary metavariables as metaid^{ty}, where *ty* indicates the *metakind* used in the declaration of the variable. For example, metaid$^{\mathsf{Type}}$ refers to a metavariable that stands for any type.

The *type* nonterminal is used by both the grammar of metavariable declarations and the grammar of transformations, and is defined on the next page.

Transformation

The grammar of the transformation is not actually the grammar of the SmPL code that can be written by the programmer, but the grammar of the slice of this consisting of the `-` annotated and the unannotated code (the context of the transformed lines), or the `+` annotated code and the unannotated code. For example, for parsing purposes, the transformation presented in Section 4.5 is split into the two variants shown below and each is parsed separately.

```
proc_info_func(...) {
  <...
  hostno
  ...>
}
```

```
proc_info_func(...) {
  <...
  hostptr->host_no
  ...>
}
```

Requiring that both slices parse correctly ensures that the rule matches syntactically valid C code and that it produces syntactically valid C code. The generated parse trees are then merged for use in the subsequent matching and transformation process.

The grammar rule for the minus or plus slice of a transformation is as follows:

transformation ::= (`#include` include_string)*
  [OPTDOTSEQ(*fun_decl_statement*$^+$ | *expr*, *stmt_whencode*)]

fun_decl_statement ::= *decl_statement* | *fun_decl*

OPTDOTSEQ(*grammar*,*whencode*) ::=
  [`...` [*whencode*]] *grammar* (`...` [*whencode*] *grammar*)* [`...` [*whencode*]]
| [`ooo` [*whencode*]] *grammar* (`ooo` [*whencode*] *grammar*)* [`ooo` [*whencode*]]
| [`***` [*whencode*]] *grammar* (`***` [*whencode*] *grammar*)* [`***` [*whencode*]]

`ooo` is analogous to `...`, but the terms may appear in any order. `***` is also analogous to `...`, but expresses interprocedural sequences. Lines may be annotated with an element of the set {`-`, `+`} or an element of the set {`!`, `?`, `\+`}, or one of each. `!`, `?`, `\+` represent exactly one, at most one, and at least one match of the given pattern. There are some constraints on the use of these annotations:

- Dots, *i.e.* `...`, `ooo`, or `***`, cannot occur on a line marked `+`.
- Nested dots, *i.e.* dots enclosed in `<` and `>`, cannot occur on a line with any marking.

Types

type ::= [`const` | `volatile`] *type_desc* (`*`)*

type_desc ::= *simple_type* | [`signed` | `unsigned`] *signable_type* | [`struct` | `union`] id
  | metaid$^{\text{Type}}$

simple_type ::= `void` | `double` | `float`

signable_type ::= `char` | `short` | `int` | `long`

Function declarations

fundecl ::= [`static`] *funid* `(` [PARAMSEQ(*param*,ε)] `)` `{` [*statement_sequence*] `}`

funid ::= *id* | metaid$^{\mathsf{Func}}$ | metaid$^{\mathsf{LocalFunc}}$

param ::= *type id* | metaid$^{\mathsf{Param}}$ | metaid$^{\mathsf{ParamList}}$

PARAMSEQ(*grammar*,*whencode*) ::=

($grammar$ `,` | ... [*whencode*] `,`)* ($grammar$ | ... [*whencode*])

| ($grammar$ `,` | `ooo` [*whencode*] `,`)* ($grammar$ | `ooo` [*whencode*])

Declarations

decl_var ::= *type* [(*id* [`[` [*dot_expr*] `]`] `,`)* *id* [`[` [*dot_expr*] `]`]] `;`

| *type id* [`[` [*dot_expr*] `]`] `=` *nest_expr* `;`

Statements

The first rule *statement* describes the various forms of a statement. The remaining rules implement the constraints that are sensitive to the context in which the statement occurs: *single_statement* for a context in which only one statement is allowed, and *decl_statement* for a context in which a declaration, statement, or sequence thereof is allowed.

statement ::= metaid$^{\mathsf{Stmt}}$

| *expr* `;`

| `if` `(` *dot_expr* `)` *single_statement* [`else` *single_statement*]

| `for` `(` [*dot_expr*] `;` [*dot_expr*] `;` [*dot_expr*] `)` *single_statement*

| `while` `(` *dot_expr* `)` *single_statement*

| `do` *single_statement* `while` `(` *dot_expr* `)` `;`

| `return` [*dot_expr*] `;`

| `{` [*statement_sequence*] `}`

| NEST(*decl_statement*$^+$ | *expr*, *stmt_whencode*)

single_statement ::= *statement* | OR(*statement*)

decl_statement ::= metaid$^{\mathsf{StmtList}}$ | *decl_var* | *statement* | OR(*statement_sequence*)

statement_sequence ::=

decl_statement* [DOTSEQ(*decl_statement*$^+$ | *expr*, *stmt_whencode*) *decl_statement**]

stmt_whencode ::= `WHEN !=` OPTDOTSEQ(*decl_statement*$^+$ | *expr*,*stmt_whencode*)

OR($grammar$) ::= `(` $grammar$ (`|` $grammar$)* `)`

DOTSEQ(*grammar*,*whencode*) ::=

... [*whencode*] (*grammar* ... [*whencode*])*

| ooo [*whencode*] (*grammar* ooo [*whencode*])*

| *** [*whencode*] (*grammar* *** [*whencode*])*

NEST(*grammar*,*whencode*) ::=

<... *grammar* (... [*whencode*] *grammar*)* ...>

| <ooo *grammar* (ooo [*whencode*] *grammar*)* ooo>

| <*** *grammar* (*** [*whencode*] *grammar*)* ***>

OR is a macro that generates a disjunction of patterns. The tokens `(`, `|`, and `)` must appear in the leftmost column, to differentiate them from the parentheses and bit-or tokens that can appear within expressions (and cannot appear in the leftmost column). These tokens are furthermore different from (, |, and), which are part of the grammar metalanguage.

Expressions

A nest or a single ellipsis is allowed in some expression contexts, and causes ambiguity in others. For example, in a sequence ... *expr* ..., the nonterminal *expr* must be instantiated as an explicit C-language expression, while in an array reference, $expr_1$ `[` $expr_2$ `]`, the nonterminal $expr_2$, because it is delimited by brackets, can be also instantiated as ..., representing an arbitrary expression. To distinguish between the various possibilities, we define three nonterminals for expressions: *expr* does not allow either top-level nests or ellipses, *nest_expr* allows a nest but not an ellipsis, and *dot_expr* allows both. The EXPR macro is used to express these variants in a concise way.

expr ::= EXPR(*expr*)

nest_expr ::= EXPR(*nest_expr*) | NEST(*nest_expr*, *exp_whencode*)

dot_expr ::= EXPR(*dot_expr*) | NEST(*dot_expr*, *exp_whencode*) | ... [*exp_whencode*]

EXPR(*exp*) ::= *exp assign_op exp* | *exp binary_op exp* | *exp* `?` [*dot_expr*] `:` *exp*

| `(` *type* `)` *exp* | *unary_op exp* | *exp* `[` *dot_expr* `]` | *exp* `.` *id*

| *exp* `->` *id* | *exp* `++` | *exp* `--` | *exp* `(` [PARAMSEQ(*arg*,*exp_whencode*)] `)`

| *id* | metaid$^{\mathsf{Func}}$ | metaid$^{\mathsf{LocalFunc}}$ | metaid$^{\mathsf{Exp}}$ | metaid$^{\mathsf{Err}}$ | metaid$^{\mathsf{Const}}$

| const | `(` *dot_expr* `)` | OR(*exp*)

arg ::= *nest_expr* | metaid$^{\mathsf{ExpList}}$

exp_whencode ::= `WHEN !=` *dot_expr*

assign_op ::= `=` | `-=` | `+=` | `*=` | `/=` | `%=` | `&=` | `|=` | `^=` | `<<=` | `>>=`

binary_op ::= `*` | `/` | `%` | `+` | `-` | `<<` | `>>` | `<` | `>` | `<=` | `>=` | `==` | `!=` | `&` | `|` | `^` | `&&` | `||`

unary_op ::= `++` | `--` | `&` | `*` | `+` | `-` | `!`

Identifiers

id ::= id | $\text{metaid}^{\text{Id}}$