Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers

Yoann Padioleau,[1] René Rydhof Hansen,[2] Julia L. Lawall,[2] Gilles Muller[1]

[1]OBASCO Group, Ecole des Mines de Nantes-INRIA, LINA 44307 Nantes cedex 3, France
[2]DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark
{Yoann.Padioleau,Gilles.Muller}@emn.fr, {rrhansen,julia}@diku.dk

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

General Terms Measurement

Keywords software evolution, domain-specific languages

1. Introduction

Developing and maintaining drivers is known to be one of the major challenges in creating a general-purpose, practically-useful operating system [1, 3]. In the case of Linux, device drivers make up, by far, the largest part of the kernel source code, and many more drivers are available outside the standard kernel source tree. New drivers are needed all the time, to give access to the latest devices. To ease driver development, Linux provides a set of driver support libraries, each devoted to a particular bus or device type. These libraries encapsulate much of the complexity of interacting with the device and the Linux kernel, and impose a uniform structure on device-specific code within a given bus or device type.

While the reliance of driver code on driver support libraries simplifies the initial development process, it can introduce long-term maintenance problems when an evolution in the library affects its interface. In this case, *collateral evolutions* are needed in all dependent device-specific code to adapt it to the new interface [19]. Simple examples of collateral evolutions include extending argument lists when a library function gets a new parameter or adjusting the context of calls to a library function when this function returns a new type of value; more complex examples involve changes that are scattered throughout a file, and where the actual code transformation is highly dependent on the specific context in which it occurs. The sheer number of device drivers and the greatly varying expertise of device driver developers and maintainers (especially in the case of driver code maintained outside the kernel source tree) has made collateral evolutions of device-specific code difficult, time-consuming, and error-prone in practice.

In order to address these problems we are developing a comprehensive language-based infrastructure, *Coccinelle*, with the goal of automating the kinds of collateral evolutions that occur in device driver code as well as providing precise, formal documentation. The features of Coccinelle are guided by the actual needs of collateral evolutions that we have found to be commonly required in Linux device driver code [19]. In this paper, we describe the initial development of Coccinelle. The main contributions are:

- The design of SmPL,[1] a language for specifying collateral evolutions relevant to device drivers; a SmPL specification serves both as detailed documentation of the collateral evolution and a concrete description of the needed code transformations.
- The design of a transformation engine for applying SmPL specifications to device driver code.
- A preliminary analysis of the effectiveness of the proposed language and transformation engine, based on an early stage prototype implementation.

The rest of this paper is organized as follows. In Section 2, we briefly summarize our previous work on understanding the scope and extent of collateral evolutions in driver code. In Section 3, we present SmPL, and in Section 4, we describe the transformation engine. Section 5 then presents some preliminary benchmarks, Section 6 presents related work and Section 7 concludes.

2. The Collateral Evolution Problem

In previous work, we have quantified various aspects of the need for collateral evolutions in Linux device drivers, using ad hoc data mining tools that we have developed [19]. These results show that driver support libraries and dependent device-specific files are numerous and the relationships between them are complex. In the Linux 2.6.13 source tree, we have identified over 150 driver support libraries and almost 2000 device-specific files. A device-specific file can use up to 59 different library functions from up to 7 different libraries. Rather than becoming more stable over time, this code base is evolving increasingly rapidly [11]. We have found that the number of evolutions in interface elements is steadily rising, as we have detected 300 probable evolutions in all of Linux 2.2 and over 1200 in Linux 2.6 up to Linux 2.6.13. Some of these evolutions trigger collateral evolutions in up to almost 400 files, at over 1000 code sites. Between Linux 2.6.9 and Linux 2.6.10, over 10000 lines of device-specific code were found to be affected by collateral evolutions.

We have also manually studied 90 of the evolutions identified during our data mining analysis. This study included examination of collateral evolutions in over 1600 device-specific files. The collateral evolutions range from simply changing the name of a function to complex transformations that involve sophisticated analysis of the usage context of each interface element, in order to construct

[1] SmPL is the acronym for “Semantic Patch Language” and is pronounced “sample” in Danish, and “simple” in French.

```
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   struct Scsi_Host *hostptr;
7
8   hostptr = scsi_host_hn_get(hostno);
9   if (!hostptr) { return -ESRCH; }
10
11  us = (struct us_data*)hostptr->hostdata[0];
12  if (!us) {
13    scsi_host_put(hostptr);
14    return -ESRCH;
15  }
16
17  SPRINTF("       Vendor: %s\n", us->vendor);
18  scsi_host_put(hostptr);
19  return length;
20 }
```

(a) Simplified Linux 2.5.70 code

```
1 static int usb_storage_proc_info (struct Scsi_Host *hostptr,
2         char *buffer, char **start, off_t offset,
3         int length, int hostno, int inout)
4 {
5   struct us_data *us;
6
7
8
9
10
11  us = (struct us_data*)hostptr->hostdata[0];
12  if (!us) {
13
14    return -ESRCH;
15  }
16
17  SPRINTF("       Vendor: %s\n", us->vendor);
18
19  return length;
20 }
```

(b) Transformed code

Figure 1. An example of collateral evolution, based on code in drivers/usb/storage/scsiglue.c

new arguments, update error handling code, etc. These collateral evolutions appear mostly to be done manually with a text editor, possibly with the help of tools such as grep. Comments in log files and mailing lists suggest that they are done by programmers with a wide range of expertise, from core Linux library maintainers, to motivated users, to developers who create and maintain drivers outside the Linux source tree. We furthermore found a number of errors that were introduced into device specific code during collateral evolutions. Many of these errors persisted for 6 months to a year, and some are still not corrected.

The results of our study clearly call for some kind of automated support for collateral evolutions. One form of automatic code updating is already widely used in the Linux community: the patch [16]. Patch code describes a specific change in a specific version of a single file. To create a patch, a developer must modify each file by hand, and then apply the diff tool to create a record of the difference between the old and new versions. The developer then distributes the patch to users, who apply it using the patch tool to replicate the changes in their copies of the old files. Although automatic at the user level, this approach does not solve the collateral evolution problem. It still requires time-consuming and error-prone manual modifications initially, and then produces an artifact that is only applicable to the files that are known to the library developer; no indication is provided how to map the collateral evolutions to drivers that are overlooked or are outside the kernel source tree. To address these issues, an approach is needed that describes collateral evolutions generically, so that they can be applied automatically, both to files inside the kernel source tree and out.

Our approach To be able to describe collateral evolutions generically, while remaining harmonious with the development model of Linux kernel code, we extend the patch notation to incorporate aspects of the semantics of C code, and not just its syntax. Accordingly, we refer to our specifications as *semantic patches*. Using semantic patches, we refine the patch model described above, such that the library developer only manually applies the collateral evolution to a few driver files, to get a feel for the changes that are required, and then writes a semantic patch that can be applied to the remaining files and distributed to other driver maintainers. While our goal is that semantic patches should apply independent of coding style, it is not possible in practice to anticipate all possible variations. Thus, the tool should not only apply semantic patches, but also be able to assist the developer or driver maintainer when an exact match of the rule against the source code is not possible.

3. SmPL in a Nutshell

In this section, we present the SmPL language for developing semantic patches through an example from our previous study [19].

The example The functions scsi_host_hn_get and scsi_host_put of the SCSI interface access and release, respectively, a structure of type Scsi_Host, and additionally manage a reference count. In Linux 2.5.71, it was decided that driver code could not be trusted to use these functions correctly, which could result in corruption of the reference count, and thus these functions were removed from the SCSI interface [15]. This evolution had collateral effects on the “proc_info” callback functions defined by SCSI drivers, which make accessible at the user level various information about the device. 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 the proc_info functions to remove the calls to scsi_host_hn_get and scsi_host_put, and to add the new argument.

Figure 1 shows a simplified version of the proc_info function of drivers/usb/storage/scsiglue.c based on that of the version just prior to the evolution, Linux 2.5.70, and the result of performing the above collateral evolutions in this function. Similar collateral evolutions were performed in Linux 2.5.71 in 19 SCSI driver files inside the kernel source tree. The affected code, shown in italics, is as follows:

- The declaration of the variable hostptr: This declaration is moved from the function body (line 6) to the parameter list (line 1), to receive the new Scsi_Host-typed argument.
- The call to scsi_host_hn_get: This call is removed (line 8), entailing the removal of the assignment of its return value to hostptr. The subsequent null test on hostptr is dropped, as the SCSI library is assumed to call the proc_info function with a non-null value.
- The calls to scsi_host_put: These calls are removed as well. Because the proc_info function should call scsi_host_put whenever scsi_host_hn_get has been called successfully (*i.e.*, returns a non-null value), there may be many such calls, one per possible control-flow path through the rest of the function. In this example, there are two: one on line 13 just before an error return and one on line 18 in the normal exit path.

```
1  @@
2  local function proc_info_func;
3  identifier buffer, start, offset, length, inout, hostno;
4  identifier hostptr;
5  @@
6    proc_info_func (
7  +       struct Scsi_Host *hostptr,
8          char *buffer, char **start, off_t offset,
9          int length, int hostno, int inout) {
10     ...
11 -   struct Scsi_Host *hostptr;
12     ...
13 -   hostptr = scsi_host_hn_get(hostno);
14     ...
15 -   if (!hostptr) { ... return ...; }
16     ...
17 -   scsi_host_put(hostptr);
18     ...
19   }
```

Figure 2. A semantic patch for updating SCSI proc_info functions

SmPL Figure 2 shows the SmPL semantic patch describing these collateral evolutions. Overall, the semantic patch has the form of a traditional patch [16], consisting of a sequence of rules each of which begins with some context information delimited by a pair of `@@`s and then specifies a transformation to be applied in this context. In the case of a semantic patch, the context information declares not line numbers but a set of *metavariables*. A metavariable can match any term of the kind specified in its declaration (identifier, expression, etc.), such that all references to a given metavariable match the same term. The transformation rule is specified as in a traditional patch file, as a term having the form of the code to be transformed. This term is annotated with the *modifiers* `-` and `+` to indicate code that is to be removed and added, respectively.

Lines 1-5 of the semantic patch of Figure 2 declare a collection of metavariables. Most of these metavariables are used in the function header in lines 6-9 to specify the name of the function to transform and the names of its parameters. Specifying the function header in terms of metavariables effectively identifies the function to transform in terms of its prototype, which is defined by the SCSI library and thus is common to all proc_info functions.Note that when a function definition is transformed, the corresponding prototype is also transformed automatically in the same way (if necessary); it is therefore not necessary to explicitly specify the transformation of a prototype in the semantic patch.

The remainder of Figure 2 specifies the removal of the various code fragments outlined above from the function body. As the code to remove is not necessarily contiguous, these fragments are separated by the SmPL operator "`...`", which matches any *sequence* of instructions. The semantic patch also specifies that a line should be added: the declaration specified in line 11 to be removed from the function body is specified to be added to the parameter list in line 7 by a repeated reference to the `hostptr` metavariable.

Overall, the rule applies independent of spacing, line breaks, and comments. Moreover, the transformation engine is parameterized by a collection of *isomorphisms* specifying sets of equivalences that are taken into account when applying the transformation rule. The default set of isomorphisms, for example, indicates that for any x that has a pointer type, `!`x, x `== NULL`, and `NULL ==` x are equivalent, and thus the pattern on line 15 of Figure 2 matches a conditional that tests the value of `hostptr` using any of these variants. A variant of the SmPL syntax is used to specify such isomorphisms. The above isomorphism is specified as follows.

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

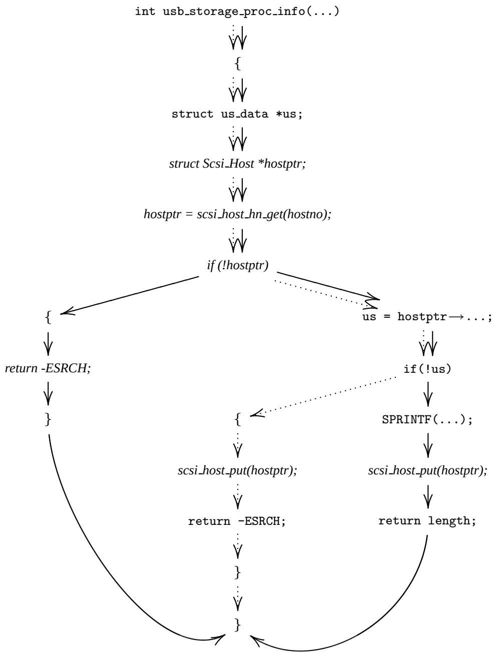

Figure 3. Control-flow graph for Figure 1 (a)

Our set of standard isomorphisms currently consists of 10 such equivalences, amounting to around 60 lines of code.

The semantics of sequences A sequence in a semantic patch represents not syntactically contiguous code, but a path in the driver's execution, *i.e.*, in its control-flow graph. For example, in the control-flow graph of Figure 3 corresponding to the program of Figure 1a, there are two paths in the right half of the graph that remain within the function after the test of `hostptr`, one represented by a solid line and one represented by a dotted line. Each path contains the function header, the declaration of `hostptr`, the call to `scsi_host_hn_get`, the null test, and its own call to `scsi_host_put` and close brace, as specified by the semantic patch. The strategy of matching within each control-flow path thus allows a semantic patch that specifies only one `scsi_host_put` to match the code of Figure 1a, which contains two, each in a separate control-flow path.

Other features SmPL contains a number of other features for matching other kinds of code patterns. These include the ability to match and transform a term wherever and however often it occurs, the ability to describe a disjunction of possible patterns, the ability to specify code that should be absent, and the ability to declare some parts of a pattern to be optional, to account for code that should be transformed if present but that may be absent either due to variations in the protocol for using the interface or due to programmer sloppiness.

4. The Coccinelle Transformation Engine

The main decision that we have taken in the design of the Coccinelle transformation engine is to base the engine on model checking technology. To this end, the C source code is translated into a control-flow graph, which is used as the model, the SmPL semantic patch is translated into a formula of temporal logic (CTL [4], with some additional features), and the matching of the formula against the model is implemented using a variant of a standard model checking algorithm [13]. This approach, which was inspired

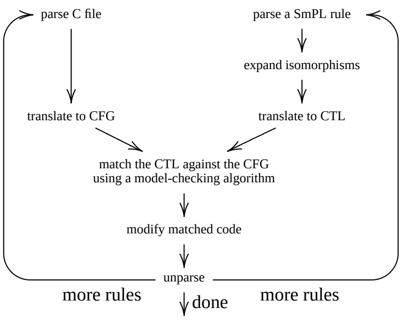

Figure 4. The Coccinelle engine

by the work of Lacey and de Moor [14] on a related but simpler transformation problem, has been crucial in rapidly developing a prototype implementation. The use of an expressive temporal logic as an intermediate language has made it possible to incrementally work out the semantics of SmPL, without affecting the underlying pattern-matching engine. Furthermore, because CTL is easy to implement, we have been able to extend the logic with two features that we have found essential to express the SmPL semantics: constructive negation [2] and existentially quantified propositional variables.

Figure 4 shows the main steps performed by the Coccinelle transformation engine, including the use of model checking. In the rest of this section, we highlight some of these steps.

Parsing the C source file A collateral evolution is just one step in the ongoing maintenance of a Linux device driver. Thus, the code generated by Coccinelle must remain readable and in the style of the original source code, to allow further maintenance and evolution. An important part of the style of the source code, which is not taken into account by most other C-code processing tools, is the whitespace, comments, and preprocessing directives. The Coccinelle C-code parser collects information about the comments and spacing adjacent to each token. When a token in the input file is part of the generated code, the associated comments and spacing are generated with it.

As has been found by others [10], parsing C code while maintaining preprocessing directives such as `#if` and `#define` and macro uses poses a significant challenge. The Coccinelle C-code parser treats all preprocessing directives as comments. Because the use of unexpanded macros may then result in code that does not follow the C grammar, we have extended the grammar accepted by the parser to address some cases that commonly occur in driver code. For example, the parser recognizes `list_for_each` as the start of a loop. While not perfect, we expect these heuristics to cover the majority of driver code. We furthermore plan to extend the tool to take into account both the unexpanded and expanded macro code, to detect collateral evolution sites in both.

Parsing the semantic patch The main issue in parsing a semantic patch is to parse the transformation rule. This consists of C-like code that is either annotated with `-` if it is to be removed, `+` if it is to be added, or not annotated if it is context code that is to be preserved by the transformation. Parsing the semantic patch as a single unit would be complex, because of the arbitrary intermingling of code annotated with `-` and `+`. Nevertheless, the code to be matched, represented by the `-` and context code, and the code to be produced, represented by the `+` and context code, should each have the form of valid C-code, modulo SmPL specific constructs, such as "`...`". We thus parse each separately using a C parser extended with the SmPL-specific constructs, and then merge the resulting ASTs so that `+` code is attached to adjacent `-` and context code. The isomorphisms are then applied to this merged AST, such that a pattern that matches any one of the set of terms designated as isomorphic is replaced by a disjunction of patterns matching the possible variants. Any `+` code associated with the subterms of such a term is propagated into all of the patterns, so that the generated code retains the coding style of the source program.

```
∃hostno, hostptr .
(∃proc_info_func, buffer, start, offset, length, inout, v .
  proc_info_func(char *buffer, char **start, off_t offset, int length, int hostno, int inout)_v)
  ∧
  AX(∃p .
      ({ ∧ Paren(p) ∧
      AX A[¬(struct Scsi_Host *hostptr; ∨ ({ ∧ Paren(p))) U
            (∃v . struct Scsi_Host *hostptr;_v ∧
             AX A[¬(hostptr = scsi_host_hn_get(hostno); ∨ struct Scsi_Host *hostptr;) U
                   (∃v . hostptr = scsi_host_hn_get(hostno);_v ∧
                    AX A[¬(scsi_host_put(hostptr); ∨ hostptr = scsi_host_hn_get(hostno);) U
                          (∃v . scsi_host_put(hostptr);_v ∧
                           AX A[¬((} ∧ Paren(p)) ∨ scsi_host_put(hostptr);) U (} ∧ Paren(p))
```

Figure 5. CTL counterpart of the semantic patch of Figure 2

The final step is to translate the merged AST into our variant of CTL. Figure 5 shows a slightly simplified CTL representation of our example (the null test on line 13 of Figure 2 and some other details are omitted for conciseness). The semantic patch consists of a sequence of fragments of the form $f\ \ \ldots\ \ g$. Such a fragment is essentially translated into:

$$f \wedge \mathsf{AX}\,\mathsf{A}[\neg(f \vee g)\ \mathsf{U}\ g]$$

meaning that first f is found in the CFG, then from all subsequent nodes in the CFG, g is eventually found, and on each path from f to g there is no occurrence of either f or g. A special existentially quantified variable v marks terms for which we want to record the matching nodes in the CFG, for subsequent transformation. The predicate *Paren* ensures that the matched braces are matching braces in the source program. For this semantic patch, the CTL translation only uses the operators conjunction, negation, AX, and AU. The translation of other SmPL operators also uses disjunction and EX. We anticipate that existential quantification over paths of arbitrary length, *i.e.* the operator EU, will be useful to express some of the collateral evolutions we have identified.

Updating the C source file The matching of the CTL formula against the control-flow graph identifies the nodes at which a transformation is required, the semantic patch code matching these nodes, and the corresponding metavariable bindings. The engine then propagates the `-` and `+` modifiers in the semantic patch code to the corresponding tokens in the matched nodes of the control-flow graph. Based on the annotated control-flow graph, the engine then generates the transformed C code. In this process, a token annotated with `-` is dropped, an unannotated token is generated as is, and a token annotated with `+` is preceded or followed as appropriate by the corresponding `+` code from the semantic patch, updated according to the metavariable environment.

In describing the parsing of the C code, we noted the need to maintain comments and spacing. The treatment of comments is especially subtle, because comments are often not contiguous to the relevant code. This makes it difficult *e.g.*, to know when all of the relevant code has been deleted, and thus the comment should be deleted as well. Currently, we keep all comments, but plan to add some heuristics to detect when comments should be removed.

5. Experiments

In this section, we describe the application of the proc_info semantic patch to Linux driver files. The transformation engine was run on a 3.2GHz Pentium 4 with 512Mb of RAM.

	file lines	proc_info fn. lines	note	seconds
block/cciss_scsi.c	1451	39		0.9
ieee1394/sbpc2.c	2985	66		3.3
scsi/53c700.c	2028	34		6.2
scsi/arm/acornscsi.c	3126	113	iso(+4)	2.8
scsi/arm/arxescsi.c	408	29		1.0
scsi/arm/cumana_2.c	574	32		0.5
scsi/arm/eesox.c	684	31		0.5
scsi/arm/powertec.c	486	32		0.8
scsi/cpqfcTSinit.c	2071	113		2.3
scsi/eata_pio.c	985	62		1.6
scsi/fcal.c	323	70		1.5
scsi/g_NCR5380.c	936	111		3.4
scsi/in2000.c	2332	153	cpp	-
scsi/ncr53c8xx.c	9481	37	iso(+6)	3.1
scsi/nsp32.c	3524	63		2.2
scsi/pcmcia/nsp_cs.c	1958	113	cpp	-
scsi/sym53c8xx.c	14738	38		9.3
scsi/sym53c8xx_2/sym_glue.c	2990	37		1.7
usb/storage/scsiglue.c	916	70		0.6

Figure 6. Experiments with the proc_info semantic patch

Some extensions were required to the semantic patch of Figure 2 to accommodate variations in driver code. We have extended the semantic patch to allow the test of the result of `scsi_host_hn_get` and the call to `scsi_host_put` to be optional. In the former case, this accounts for the fact that error checking is not uniformly done in the Linux kernel. In the latter case, this accounts for the fact that the call to `scsi_host_put` is often omitted, which was indeed the motivation for the evolution. The semantic patch is also extended with some other transformations that were performed as part of the same set of collateral evolutions. The resulting semantic patch is 44 lines of code. Three of the ten standard isomorphisms apply to this semantic patch.

Figure 6 lists the files affected by the proc_info collateral evolutions, the number of lines of code in the complete file, the number of lines of code in the proc_info function, and the time required to transform the file. Application of the semantic patch is fully automated for 15 out of the 19 relevant driver files. For two of the remaining files, noted “iso”, some minor additions to the semantic patch were required to simulate isomorphisms that have not yet been implemented in the general case; the number of lines manually added is shown in parenthesis. Finally, the two remaining files, noted “cpp”, depend on the C preprocessor in ways that our prototype does not yet handle. We are working on these issues.

The transformation time is dominated by the time to parse the file and the time to treat the proc_info functions, as other functions are immediately rejected by the transformation rule. For smaller files this completes in a few seconds, while the larger files that we have successfully transformed require fewer than 10 seconds.

6. 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 [17], 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 [18], which allows specifying patterns using metavariables and provides some isomorphisms, and the work of De Volder on JQuery [6], 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 [14].

Other work. Refactoring is a generic program transformation that reorganizes the structure of a program without changing its semantics [9]. Some of the collateral evolutions in Linux drivers can be seen as refactorings. Refactorings, as originally designed, however, apply to the whole program, requiring access 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. Henkel and Diwan have also observed that refactoring does not address the needs of evolution of libraries when the client code is not available to the library maintainer [12]. Their tool, CatchUp, can record some kinds of refactorings and replay them on client files. Nevertheless, CatchUp is only implemented in the Eclipse IDE and only handles a few of the refactorings provided by Eclipse. We have furthermore found that many collateral evolutions are specific to the OS API, and thus cannot be described as part of a generic refactoring.

The JunGL scripting language allows programmers to implement new refactorings [22]. Although this language should be able to express collateral evolutions, a JunGL transformation rule does not follow the structure of the source terms, and thus does not make visually apparent the relationship between the code fragments to be transformed. We have found that this 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.

Coady et al. have used Aspect-Oriented Programming (AOP) to extend OS code with new features [5, 8]. Nevertheless, AOP is targeted towards modularizing concerns rather than integrating them into a monolithic source code. In the case of collateral evolutions, our observations, *e.g.* of the limited use of wrapper functions, suggest that Linux developers favor approaches that update the source code, resulting in uniformity among driver implementations.

Analysis tools in Linux. The Linux community has recently begun using various tools to better analyze C code. Sparse [20] 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 [21] 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. [7] 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.

7. Conclusion

In this paper, we have proposed a declarative language, SmPL, for expressing semantic patches and presented the design of a transformation engine for applying these semantic patches to driver code. SmPL is based on the patch syntax familiar to Linux developers, but enables transformations to be expressed in a more general form. The transformation engine is defined in terms of control flow rather than syntactic structure and is configurable by a collection of isomorphisms, so that a single semantic patch can be applied to drivers exhibiting a variety of coding styles.

Preliminary experiments with our prototype implementation have shown promising results. We have been able to implement a concise semantic patch describing the collateral evolutions of proc_info functions, and to successfully apply this semantic patch to most of the affected files. In many cases, the transformation is performed in under 2 seconds. We are currently developing other semantic patches, targeting collateral evolutions found in our previous study. Our initial results show that it is possible to write concise semantic patches that express these transformations as well.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 73–85, Leuven, Belgium, Apr. 2006.

[2] R. Bártak. Constructive Negation CLP(H). Technical Report 98/6, Department of Theoretical Computer Science, Charles University, Prague, Czech Republic, July 1998.

[3] 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.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] A. Garrido. *Program refactoring in the presence of preprocessor directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[11] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *International Conference on Software Management (ICSM)*, pages 131–142, 2000.

[12] J. Henkel and A. Diwan. CatchUp! capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th international conference on Software engineering*, pages 274–283, St. Louis, MO, USA, May 2005.

[13] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[14] 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.

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

[16] 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.

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

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

[19] 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.

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

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

[22] 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.