Documenting and Automating Collateral Evolutions in Linux Device Drivers

Yoann Padioleau
Ecole des Mines de Nantes
yoann.padioleau@acm.org

Julia Lawall
DIKU, University of Copenhagen
julia@diku.dk

René Rydhof Hansen
Aalborg University
rrh@cs.aau.dk

Gilles Muller
Ecole des Mines de Nantes
Gilles.Muller@emn.fr

ABSTRACT

The internal libraries of Linux are evolving rapidly, to address new requirements and improve performance. These evolutions, however, entail a massive problem of *collateral evolution* in Linux device drivers: for every change that affects an API, all dependent drivers must be updated accordingly. Manually performing such collateral evolutions is time-consuming and unreliable, and has lead to errors when modifications have not been done consistently.

In this paper, we present an automatic program transformation tool, Coccinelle, for documenting and automating device driver collateral evolutions. Because Linux programmers are accustomed to manipulating program modifications in terms of patch files, this tool uses a language based on the patch syntax to express transformations, extending patches to *semantic patches*. Coccinelle preserves the coding style of the original driver, as would a human programmer.

We have evaluated our approach on 62 representative collateral evolutions that were previously performed manually in Linux 2.5 and 2.6. On a test suite of over 5800 relevant driver files, the semantic patches for these collateral evolutions update over 93% of the files completely. In the remaining cases, the user is typically alerted to a partial match against the driver code, identifying the files that must be considered manually. We have additionally identified over 150 driver files where the maintainer made an error in performing the collateral evolution, but Coccinelle transforms the code correctly. Finally, several patches derived from the use of Coccinelle have been accepted into the Linux kernel.

> *"The Linux USB code has been rewritten at least three times. We've done this over time in order to handle things that we didn't originally need to handle, like high speed devices, and just because we learned the problems of our first design, and to fix bugs and security issues.* ***Each time we made changes in our api, we updated all of the kernel drivers that used the apis, so nothing would break.*** *And we deleted the old functions as they were no longer needed, and did things wrong."* Greg Kroah-Hartman, OLS 2006.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Languages, Reliability, Measurement

Keywords

Linux, device drivers, software evolution, collateral evolutions, program transformation, domain-specific languages

1. INTRODUCTION

Evolution in systems software is essential, to improve performance, enhance security, and support new hardware. Despite these benefits, however, evolution can also reduce code quality, as an evolution that affects a library API can break all code that depends on it. In this case, *collateral evolutions* are needed in all dependent code to update it accordingly. Collateral evolutions may range from changing the name of a single library function at every call site to making multiple distinct context-dependent changes throughout the affected files.

In previous work, we have studied the problem of collateral evolutions in the context of Linux device drivers [20]. Drivers make up around 50% of the Linux kernel source tree, they rely heavily on Linux internal library APIs, and their correctness is essential to the usability of the OS. Currently, collateral evolutions in driver code are typically performed manually or using unstructured and error-prone regular expressions with tools such as `sed` or Perl. Manual modifications are tedious and time-consuming, and regular expressions are difficult to write for complex changes. In some cases, the collateral evolution process has taken several years [20]. Furthermore, while the developer who updates a library often performs the needed collateral evolutions on the drivers in the Linux source tree, many drivers are maintained outside the kernel source tree by device experts or by users who may not be familiar with the subtleties of the evolutions in the internal libraries. Not surprisingly, bugs have been introduced in performing collateral evolutions, ranging from simple typing errors to semantic misunderstandings [20].

In this paper, we present a transformation tool, Coccinelle, for documenting and automating device driver collateral evolutions. To be useful to library developers and driver maintainers, this tool must meet the following challenges:

Ease of use. Library developers and driver maintainers are often not familiar with program transformation tools. To gain acceptance, a program transformation tool must thus fit with the notations and processes familiar to those working on Linux code.

Preservation of coding style. Because a collateral evolution is just one step in the ongoing maintenance of a driver, transformed driver code must follow the same coding style as the original. This includes preserving the use of macros and C preprocessor directives, which are notoriously difficult to handle [7, 16].

Genericity. A transformation rule describing a collateral evolution must be applicable to a wide range of drivers, including those outside the Linux source tree, which may not be available to the library developer. Thus, transformation rules must be independent of irrelevant code variations in device-specific code. Moreover, the C language allows some operations, such as null pointer checks, to be expressed in multiple ways. Transformation rules must also be insensitive to these variations.

Efficiency. Recent versions of Linux include over 4000 driver files, and some collateral evolutions that affect drivers affect other kernel services as well. A transformation tool must be efficient enough to allow interactive use, even when applied to the entire Linux source tree.

Collateral evolutions in Linux device drivers, however, have some properties that help address these challenges. First, Linux developers already exchange code transformations in terms of a specific formal notation, the patch file [14]. This notation can thus provide the basis for a language for expressing program transformations that is compatible with the current habits of Linux developers. Second, there are efforts to standardize the use of macros in Linux code [23]. Thus, it is tractable to directly parse driver code containing preprocessor directives in most cases. Third, the structure of the code that uses API functions is largely dictated by the constraints imposed by the library, and thus is mostly impervious to coding style. Indeed, many drivers are written by copy-paste from an existing driver [12]. Finally, many APIs are used in a fairly localized way, making it possible to efficiently process even large driver files.

Building on these properties of driver code, we have developed the Coccinelle transformation tool. The main contributions of our work are as follows:

- A WYSIWYG approach to describing collateral evolutions in terms of *semantic patches*, which like traditional patches describe transformations using fragments of ordinary C code. Semantic patches are thus easy to create from sample driver source code, and easy for other driver maintainers to read and understand.

- A parser that is able to accommodate many uses of C preprocessing (`cpp`) directives directly. Comments and whitespace are maintained where possible in the transformed code to further preserve the ability to understand and maintain the driver.

- A notion of *isomorphisms*, which equate semantically equivalent code fragments, to abstract away from variations in coding style, and the use of temporal logic to abstract away from variations in device-specific execution paths. The resulting genericity implies that, as shown in Section 5, a single semantic patch of under 50 lines can suffice to update over 500 files.

- Strategies to optimize the transformation process such that the average treatment time per affected driver is 0.7 seconds and the entire Linux kernel can be processed in less than 1 minute.

- An evaluation of Coccinelle on a range of over 60 typical collateral evolutions performed in earlier versions of Linux. On this test suite, Coccinelle correctly updates over 93% of the files previously identified by the human programmer.

- A preliminary evaluation of the use of Coccinelle to complete collateral evolutions that were only incompletely performed on Linux code, as indicated by the Linux kernel janitors mailing list and other similar sources. 24 patches derived from the use of Coccinelle have been accepted into Linux.

The rest of this paper is organized as follows. Section 2 presents current collateral evolution practice and how Coccinelle aids in this process. Section 3 presents our language SmPL for specifying semantic patches and Section 4 presents the associated transformation engine. Section 5 evaluates Coccinelle in terms of a variety of representative examples, Section 6 describes some limitations of the approach, and Section 7 describes our contribution to Linux. Finally, Section 8 presents related work, and Section 9 concludes.

2. USING COCCINELLE

Collateral evolution in Linux device drivers typically proceeds as follows: When a *library developer* makes a change in an internal Linux library that has an impact on the API, he manually updates all of the relevant device-specific code in the Linux source tree, based on his understanding of the evolution and his familiarity with the affected drivers. Next, the *maintainers* of the many drivers outside the Linux source tree perform the collateral evolution in their own code. These driver maintainers do not have first-hand knowledge of the evolution, and thus must infer how it applies to their code from any available code comments, informal mailing list discussions, and often voluminous patches. Finally, *motivated users* apply the collateral evolutions to code that was overlooked by the library developer or driver maintainer. A motivated user may have no previous experience with either the evolution or the driver code, and thus the problems faced by the driver maintainer are compounded in this case. We now consider how Coccinelle can fit into the various aspects of this collateral evolution process.

Coccinelle and the library developer. When modifying a library, a library developer can use Coccinelle to reduce the time needed for manual code modifications, to improve the robustness of the collateral evolution process, and to provide formal documentation of the changes that are required. To this end, a developer who modifies a library also

```
static int usb_storage_proc_info (
        char *buffer, char **start, off_t offset,
        int length, int hostno, int inout)
{
   struct us_data *us;
   struct Scsi_Host *hostptr;

   hostptr = scsi_host_hn_get(hostno);
   if (!hostptr) { return -ESRCH; }

   us = (struct us_data*)hostptr->hostdata[0];
   if (!us) {
     scsi_host_put(hostptr);
     return -ESRCH;
   }

   SPRINTF("       Vendor: %s\n", us->vendor);
   scsi_host_put(hostptr);
   return length;
}
```

(a) Simplified Linux 2.5.70 code

```
static int usb_storage_proc_info (struct Scsi_Host *hostptr,
        char *buffer, char **start, off_t offset,
        int length, int hostno, int inout)
{
   struct us_data *us;





   us = (struct us_data*)hostptr->hostdata[0];
   if (!us) {

     return -ESRCH;
   }

   SPRINTF("       Vendor: %s\n", us->vendor);

   return length;
}
```

(b) Transformed code

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

writes a semantic patch that describes the entailed collateral evolutions. As presented in the next section, Coccinelle uses a C-like notation for semantic patches, which means that the developer can often start with a typical driver, perform the transformation by hand, apply `diff` to the old and new versions to create a standard patch and then edit this patch to abstract away from inessential details. To test his intuitions and bring the affected drivers up to date, the developer applies the semantic patch across the kernel source tree. If this initial semantic patch does not apply completely to all of the affected drivers, *e.g.*, due to unanticipated variations in coding style, then he must iteratively refine and test it. To aid in this process, Coccinelle provides an interactive mode based on Emacs' `ediff` where the developer can approve each change before it is applied to the source code, and gives information about cases where the semantic patch only partially matches the source code. Eventually, the library developer is confident that the semantic patch addresses all relevant cases. He can then apply it one final time to the drivers in the Linux kernel source tree, with the assurance that changes will be made uniformly in all relevant files, and then publish it for use by driver maintainers in a repository analogous to the current repository for standard Linux patches,[1] providing a complete formal record of the acquired expertise.

Coccinelle and the driver maintainer or motivated user. When faced with a driver that is not compatible with a recent version of the Linux kernel, a programmer can search the repository of semantic patches for those corresponding to collateral evolutions between the Linux version supported by his driver and the desired one, and apply them to his driver. This raises the issue, however, of whether a semantic patch that was created without knowledge of his code will apply correctly. Due to the copy-paste model of development and the constraints on the use of the API, the semantic patch is likely to be compatible with the driver code. Still, the programmer can first read the semantic patch, which identifies in terms of C code fragments the complete set of code patterns that are affected by the collateral evolution, and compare them to the structure of his driver. Next, he can apply the semantic patch using the Coccinelle interactive mode. If Coccinelle indicates that the semantic patch only partially matches the driver code, he can refine the semantic patch, following the same iterative process as the library developer, or simply modify the affected code by hand, using the semantic patch as a formal guideline. Finally, he may contribute a refined semantic patch to the repository.

[1]http://git.kernel.org/

Summary. The benefits of Coccinelle for both a library developer and a driver maintainer are the closeness of the semantic patch language to C code, the automatic transformation tool that makes it possible to easily and reliably apply transformations to driver code, and the feedback provided by the interactive usage mode and the partial matches. We describe and assess these features in the rest of this paper.

3. SMPL IN A NUTSHELL

Coccinelle provides the language SmPL (Semantic Patch Language) for writing semantic patches. The design of SmPL is guided by the goal of providing a declarative, WYSIWYG approach that is close to the C language and builds on the existing patch notation. 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.

The proc_info collateral evolution. The proc_info collateral evolution concerns the use of the SCSI API functions `scsi_host_hn_get` and `scsi_host_put`, which 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, 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 [13]. This evolution had collateral effects on the "proc_info" callback functions[2] defined by SCSI drivers, which call these API functions. 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 a `Scsi_Host`-typed structure as an argument

[2]A proc_info callback function makes accessible at the user level various information about the device.

```
1  @ rule1 @
2  struct SHT ops;
3  identifier proc_info_func;
4  @@
5    ops.proc_info = proc_info_func;
6
7  @ rule2 @
8  identifier buffer, start, offset, length, hostno, inout;
9  identifier hostptr, rule1.proc_info_func;
10 @@
11   proc_info_func (
12 +       struct Scsi_Host *hostptr,
13         char *buffer, char **start, off_t offset,
14         int length, int hostno, int inout) {
15     ...
16 -   struct Scsi_Host *hostptr;
17     ...
18 -   hostptr = scsi_host_hn_get(hostno);
19     ...
20 -   if (!hostptr) { ... return ...; }
21     ...
22 -   scsi_host_put(hostptr);
23     ...
24   }
```

Figure 2: A semantic patch for performing the proc_info collateral evolutions. Metavariables are shown in italics.

to these callback functions. 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 the version just prior to the evolution, from 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:

The declaration of the variable `hostptr`, which 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`, which 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 provide a non-null value.

The calls to `scsi_host_put`, which are removed. 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. 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.

The proc_info collateral evolution using SmPL. Figure 2 shows a SmPL semantic patch describing these collateral evolutions. Overall, the semantic patch has the form of a traditional patch [14], 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 a set of *metavariables.* A metavariable can match any term of the kind specified in its declaration (identifier, integer expression, etc.), such that all references to a given metavariable match the same term. The transformation 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. The semantic patch in Figure 2 consists of two rules: `rule1` (lines 1-5) finds the name of the proc_info function in the given SCSI driver file, and `rule2` (lines 7-24) specifies the transformation that should be applied to the definition of that function.

`Rule1` declares two metavariables (lines 1-4): *ops* and *proc_info_func*. The metavariable *ops* is declared as an arbitrary expression of type `struct SHT`, and represents the structure that a SCSI driver passes to the SCSI library to identify the driver's callback functions. The metavariable *proc_info_func* is declared as an identifier and represents the name of the proc_info function. The rest of this rule (line 5) simply identifies the proc_info function as the function that is stored in the `proc_info` field of the structure *ops*. For example, the scsiglue driver stores the function `usb_storage_proc_info` in this field (code not shown). `Rule1` can match any number of times in a given driver file. The rest of the semantic patch will be applied once for each possible distinct set of bindings of the metavariables.

`Rule2` declares metavariables to represent the parameters of the proc_info function and the `Scsi_Host`-typed local variable (lines 7-10). The metavariable *proc_info_func* is specified to be *inherited* from `rule1`. The rest of the rule (lines 11 to 24) specifies the removal of the various code fragments outlined above from the function body. Because 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 16 to be removed from the function body is specified to be added to the parameter list in line 12 by a repeated reference to the `hostptr` metavariable. These transformations only apply if the rule matches completely.

Abstracting away from syntactic variations. A semantic patch applies independent of spacing, line breaks, and the presence of comments. For example, the semantic patch of Figure 2 matches the code of Figure 1(a), even though the opening brace of the function definition is on the same line as the function header in the semantic patch (line 14 of Figure 2) and on a different line in the code (line 4 of of Figure 1(a)). Moreover, the Coccinelle transformation engine is parametrized by a collection of *isomorphisms* specifying sets of equivalences that are taken into account when applying the transformation rule. Isomorphisms can be specified in an auxiliary file by the SmPL programmer, using a variant of the SmPL syntax. Among the default set of isomorphisms is the property that for any x that has pointer type, `!`x, x `== NULL`, and `NULL ==` x are equivalent. This isomorphism is specified as follows:

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

Given this specification, the pattern on line 20 of Figure 2 matches a conditional that tests `hostptr` using any of the listed variants. In `rule1`, isomorphisms account for the various ways of initializing a structure field (line 5), *i.e.*, via the structure itself, via a pointer to the structure, or using the C99 initializer syntax. In `rule2`, isomorphisms also allow the local variable **hostptr** to be initialized to a constant at the point of its declaration (line 16) and allow the braces around the then branch of the test of the result of calling **scsi_host_hn_get** to be omitted (line 20). All of these isomorphisms are exploited in transforming the relevant drivers in the Linux kernel source tree. Our default set of isomorphisms contains 39 equivalences commonly relevant to driver code.

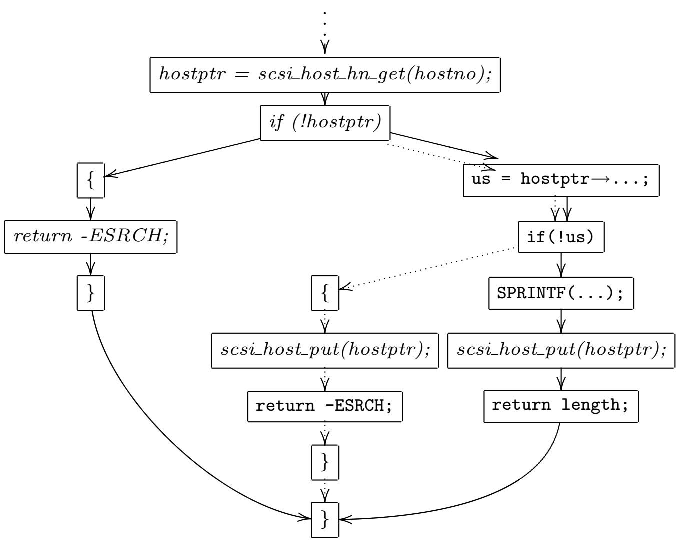

Figure 3: CFG for Figure 1, lines 9-20 (a)

Abstracting away from control-flow variations. A device driver implements an automaton, testing various conditions depending on the input received from the kernel and the current state of the device, in order to determine an appropriate action. Thus, a driver function is typically structured as a tree, with many exit points. An example is the code in Figure 1a, which contains branches on lines 9 and 12. As the structure of this tree depends on device-specific properties, it cannot be explicitly represented in a semantic patch. Instead, the semantic patch describes the pattern of runtime operations required by the library, which should be the same across all drivers.

As an approximation of the pattern of operations performed at run time by a driver, we use paths in the driver's control-flow graph (CFG). Thus, a sequence of terms in a SmPL semantic patch matches a sequence of C-language terms along such a path. In particular, the SmPL construct "`...`" represents an arbitrary CFG path fragment. The need for this approach is illustrated by the scsiglue proc_info function of Figure 1a, for which the CFG is shown in Figure 3. This function contains two calls to **scsi_host_put**, while the semantic patch of Figure 2 contains only one. The right side of the CFG of Figure 3 shows that there are two paths that remain within the function after the test of **hostptr**, one represented by a solid line and one represented by a dotted line. Considering the entire execution of the function, each path consists of 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 **rule2** of the semantic patch. Thus, the semantic patch matches this code. The semantic patch furthermore matches

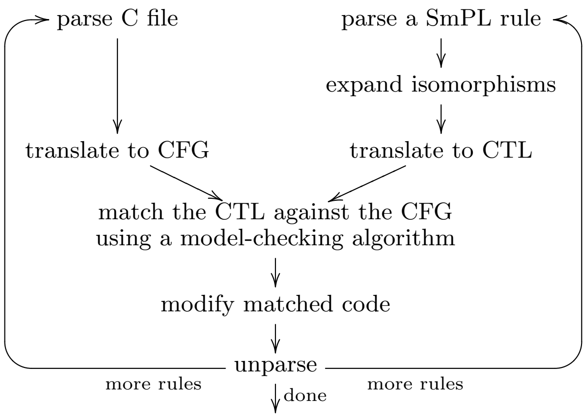

Figure 4: The Coccinelle engine

proc_info functions having any other number of branches and a corresponding number of calls to `scsi_host_put`.

Other features. SmPL contains a number of other features for matching other kinds of code patterns. These include the ability to describe a disjunction of possible patterns to be tried in order, the ability to specify code that should not be present, and the ability to declare some parts of a pattern to be optional. Semantic patches may furthermore use almost all of C and many `cpp` constructs (`#define`, etc.), and use metavariables abstracting over all kind of terms, including identifiers, constants, expressions of various types, statements, and parameter lists. All of these features are illustrated in the examples in the appendix. These features allow a semantic patch to match over and transform almost any kinds of constructs, which is essential because of the wide range of transformations required for collateral evolutions.

4. THE COCCINELLE ENGINE

In our analysis of the collateral evolution problem, we identified the need for an approach that fits with the habits of Linux programmers, that can preserve coding style, that can accommodate code variations, and that is efficient. The language SmPL satisfies the first requirement, as it builds on the Linux patch format. The challenge is then to implement this language so as to satisfy the remaining requirements. In this section, we present the Coccinelle transformation engine, that takes as input a semantic patch and a driver, and transforms the driver according to the semantic patch. Figure 4 shows the main steps performed by the engine. The key points in its design are the strategies for (i) coping with `cpp`, to be able to preserve the original coding style, (ii) abstracting away from syntactic and control-flow variations, and (iii) efficiently applying code transformations. The implementation of Coccinelle amounts to over 60,000 lines of OCaml code.

Coping with `cpp`. Because collateral evolutions are just one step in the ongoing maintenance of a Linux device driver, the Coccinelle engine must generate code that is readable and in the style of the original source code, including the use of cpp constructs. Furthermore, a collateral evolution may involve cpp constructs. Therefore, Coccinelle parses C code as is, without expanding cpp preprocessing directives, so that they are preserved in both the transformation process and the generated code.

It is non-trivial to parse C code that contains preprocessing directives [7, 16]. While many macro uses can be parsed as an identifier reference or a function call, others, such as uses of the widely used list_for_each macro which expands to a loop header, do not have the form of a valid C term. Furthermore, conditional compilation directives such as #ifdef can break the term structure, as illustrated by the following:

```
#if LINUX_VERSION_CODE >= 0x010346
int gdth_reset(Scsi_Cmnd *scp, int reset_flags) {
#else
int gdth_reset(Scsi_Cmnd *scp) {
#endif
```

We have developed a number of heuristics that, *e.g.*, use indentation to identify macro uses that represent loop headers, and that recognize common patterns of conditional compilation, such as the duplicated function header illustrated above. These heuristics allow us to parse 99% of the code in the recent Linux v2.6.21 kernel, and 99% of the code of the over 5800 driver files in our test suite drawn from Linux 2.5 and 2.6. The remaining parsing problems are due to real syntax errors, to code containing patterns not yet handled by our heuristics, and to code for which it seems very difficult to propose heuristics. In the last case, because of the small number of lines of code involved, we hope to convince the Linux community to rewrite the code.

To preserve the original coding style, our C-code parser also collects information about the whitespace and comments adjacent to each token. When a token in the input file is part of the generated code, the associated whitespace and comments are generated with it in the unparsing phase.

Abstracting away from syntactic and control-flow variations. Abstracting away from syntactic variations is implemented using the isomorphisms. Initially, a semantic patch is parsed, to create a pattern consisting of the minus and unannotated code, which is then decorated by the plus code at the points at which this code should be inserted. The isomorphisms are then applied to this pattern. Any subpattern that matches any one of the set of terms designated as isomorphic is replaced by a disjunction of patterns matching the possible variants. The + 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. As an example, given the null pointer testing isomorphism described in Section 3, the pattern $\texttt{x}^{-+\texttt{y}}$ == NULL, in which x is decorated by an indication that it should be replaced by y, would be converted to a disjunction of the patterns $\texttt{x}^{-+\texttt{y}}$ == NULL, $!\texttt{x}^{-+\texttt{y}}$, and NULL == $\texttt{x}^{-+\texttt{y}}$. This disjunction replaces x by y in whatever form the null test occurs.

SmPL abstracts away from control-flow variations by taking the strategy of matching a sequence in a semantic patch against a path in the matched code's control-flow graph. To reason about control-flow graphs, we have based the Coccinelle engine on model checking technology, which is naturally graph-based [10]. In this approach, each top-level block of C source code (*e.g.*, a function or macro definition) is translated into a control-flow graph, which is used as the model, and the SmPL semantic patch is compiled into a formula of temporal logic (CTL [2], with some additional features). The matching of the formula against the model is then implemented using a variant of a standard model checking algorithm [10]. While CTL is probably unfamiliar to most driver maintainers, it serves only as an assembly language, which driver maintainers do not have to read or understand. We have found the use of an expressive logic such as CTL to be crucial in rapidly developing a prototype implementation, as it has allowed us to incrementally work out the semantics of SmPL, without affecting the underlying pattern-matching engine.

The result of matching the CTL formula against the control-flow graph is a collection of nodes where a transformation is required, the fragment of the semantic patch that matches these nodes, and the metavariable bindings. The engine then propagates the - and + annotations on the semantic patch to the corresponding tokens in the matched nodes of the control-flow graph, and from there to the abstract-syntax tree. The engine then unparses the abstract-syntax tree, dropping any token annotated with - and adding the + code from the semantic patch, as appropriate. Any metavariables in the added + code are instantiated according to the bindings returned by the CTL matching process.

Efficiency. The rule-matching process, including the model checking, is reasonably efficient. Nevertheless, applying the complete process to every file in the Linux source tree would be expensive, and unnecessarily so, because there are typically many files that are not relevant to a given collateral evolution. Thus, Coccinelle provides several optimizations to quickly identify relevant files and functions. At a coarse grain, the SmPL compiler collects key words, such as function and field names, that must appear in a file for the semantic patch to apply. For instance, for the semantic patch of Figure 2, these are the proc_info field and the names of the scsi_host_hn_get and scsi_host_put functions. The Coccinelle engine then uses grep, or the full-text indexing and search tool glimpse [?] if this option was activated by the user, to identify files that contain at least one of these key words; other files are not even parsed. At a more fine grain, for each rule, the SmPL compiler also collects atomic patterns, such as function calls, that must appear in a function for the rule to be applicable. Checking for these patterns is done after parsing and CFG creation, but if some required pattern is not present, the model checker is not invoked. These optimizations make it possible to apply even complex semantic patches to the entire Linux kernel source tree in a reasonable amount of time (details in Section 5).

5. EXPERIMENTS

In this section, we evaluate the application of Coccinelle to driver code, illustrating 62 collateral evolutions, in a wide range, from simple to complex, and from highly specialized to generic. First, we consider our running example, and describe some extensions to the semantic patch presented in Section 3 and its performance on the Linux kernel. Second, we consider collateral evolutions that apply at a very large number of sites in driver code. Third, we consider collateral evolutions that have proved difficult for developers, as evidenced by the errors that have been introduced. Finally, we evaluate the complete set of collateral evolutions affecting drivers in the bluetooth directory since Linux 2.6.12. Overall, these examples affect over 5800 driver files, from Linux 2.5 and 2.6. Our test machine is a 3.4GHz Pentium 4 PC with 1024MB of RAM.

5.1 Methodology

In each of our experiments, we identified the relevant collateral evolutions, wrote the semantic patches, applied these semantic patches to driver code, and finally assessed the correctness of the results. In this, we were helped by some recent developments in Linux patch management. The Linux documentation on submitting patches requests that each patch contain only one logical change, and this advice has been largely followed in the recent patches we have studied. It is thus generally possible to distinguish the patches containing collateral evolutions from the ones containing device-specific changes. Furthermore, since version 2.6.12, Linux has used the `git` version control system, which permits extracting the versions of the various driver files immediately before the application of any patch since Linux 2.5.0.[3] This allows us to test the semantic patches without interference from other changes.

To identify collateral evolutions, we have applied our tool `patchparse` [20] to Linux patches, to detect commonly occurring changes. From this information and study of the affected drivers, we have manually identified related changes, which we have then implemented as a semantic patch. To test the semantic patch, we have applied it to each of the driver files in the patches identified by `patchparse`, in their state just before the application of the collateral evolution, and compared the result to the result generated by the (traditional) patch file. We have ignored whitespace and the contents of comments, as well as any device-specific changes, for patches that do not completely follow Linux policy.

5.2 Running example

We begin with the semantic patch presented in Section 3. As compared to the simplified version of Figure 2, some extensions to the semantic patch are required to complete the proc_info collateral evolution, in practice. With respect to the code in Figure 2, we annotate the testing of the return value of the call to `scsi_host_hn_get` and the call to `scsi_host_put` as optional, as these are not present in all drivers (indeed the frequent omission of `scsi_host_put` was the motivation for the collateral evolution). Some further minor additions were required to simulate isomorphisms that have not yet been implemented in the general case. As shown in Figure 5, we also extend the semantic patch with two new rules, `rule3` and `rule4`. `Rule3` (lines 1-9) replaces the use of the `hostno` parameter by a field access from the new `hostptr` parameter, as required by the collateral evolution. In this rule, the SmPL operators `<...` and `...>` enclose a term which is matched and transformed wherever and however often it occurs, analogous to the `/g` modifier of `sed`. Finally, `rule4` (lines 11-23) adjusts any local calls to the `proc_info` function. The resulting semantic patch is 63 lines of code, less than one tenth the size of the (traditional) patch for these files (692 lines).

Figure 6 lists the 19 files in the Linux kernel source tree affected by the proc_info collateral evolution, the number

```
1  @ rule3 @
2  identifier rule1.proc_info_func, rule2.hostno, rule2.hostptr;
3  @@
4    proc_info_func(...) {
5      <...
6  -   hostno
7  +   hostptr->host_no
8     ...>
9    }
10
11 @ rule4 @
12 identifier func, hostptr, rule1.proc_info_func;
13 expression buffer, start, offset, length, inout, hostno;
14 @@
15  func(..., struct Scsi_Host *hostptr, ...) {
16   <...
17    proc_info_func(
18 +        hostptr,
19          buffer, start, offset, length,
20 -        hostno,
21          inout)
22    ...>
23  }
```

Figure 5: The remaining proc_info rules

of lines of code in each file, the number of lines of code in each proc_info function, and the time required to transform each file. The semantic patch applies in less than a second in almost all cases, regardless of the size of the file or the proc_info function, and in less than 2 seconds in the worst case. Furthermore, running Coccinelle on all the 5838 `.c` files in the image of Linux from just before the collateral evolution takes 2 minutes, and correctly updates the 19 relevant driver files. Coccinelle can also take advantage of index information, as calculated by `glimpse` [?]. If this information is used, it takes only 50 seconds to apply the semantic patch to the whole kernel.

	file LOC	proc_info LOC	Running time
block/cciss_scsi.c	1451	39	0.2s
ieee1394/sbp2.c	2985	66	0.6s
scsi/53c700.c	2028	34	0.3s
scsi/arm/acornscsi.c	3126	113	0.4s
scsi/arm/arxescsi.c	408	29	0.2s
scsi/arm/cumana_2.c	574	32	0.1s
scsi/arm/eesox.c	684	31	0.1s
scsi/arm/powertec.c	486	32	0.2s
scsi/cpqfcTSinit.c	2071	113	0.4s
scsi/eata_pio.c	985	62	0.2s
scsi/fcal.c	323	70	0.3s
scsi/g_NCR5380.c	936	111	0.9s
scsi/in2000.c	2332	153	1.9s
scsi/ncr53c8xx.c	9481	37	0.9s
scsi/nsp32.c	3524	63	0.4s
scsi/pcmcia/nsp_cs.c	1958	113	0.5s
scsi/sym53c8xx.c	14738	38	1.6s
scsi/sym53c8xx_2/sym_glue.c	2990	37	0.4s
usb/storage/scsiglue.c	916	70	0.2s

Figure 6: Experiments with proc_info

5.3 Mega-collateral evolutions

While some driver support libraries are highly specialized, others are used by many kinds of drivers spread across the Linux source tree. Evolutions in such libraries may entail "mega" collateral evolutions, affecting up to hundreds of files at thousands of code sites. When such collateral evolutions are slightly complex, they often exceed the capacity of a

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

	Manual evolution					Coccinelle				
Evolution	Files	patch LOC (change LOC)	Maint.	Duration	Miss & Err.	SP LOC	Patch/ SP LOC	Running time Avg. (Max.)	Miss	% OK
M1. rename function `pci_module_init`	180	3171 (400)	21	2 years	0	6	528x	0.4s (2.3s)	1	99%
M2. eliminate the `pt_regs` parameter	931	30083 (5467)	13	3 months	39	407	74x	1.1s (6.9s)	25	97%
M3. replace `kmalloc/memset` by `kzalloc`	556	22789 (4739)	47	1 year	25	309	74x	1.8s (90.0s)	56	90%
M4. `constify file_operations` structures	140	2569 (426)	4	5 months	2	48	54x	0.2s (0.9s)	0	100%
M5. rename type `termios`	100	2624 (566)	1	1 day	0	154	17x	0.5s (2.6s)	23	77%
M6. add `path` substructure	56	1779 (314)	1	1 day	0	16	111x	0.2s (0.9s)	0	100%
M7. change `if()BUG();` to `BUG_ON();`	82	2323 (517)	3	9 months	5	6	387x	0.3s (1.8s)	0	100%
M8. use `ARRAY_SIZE` macro	171	5407 (1099)	9	1 year	8	115	47x	1.0s (17.9s)	17	90%
M9. drop `\#include<linux/config.h>`	1124	44580 (2978)	15	1 year	0	2	22290x	0.2s (2.8s)	35	97%
M10. remove `Scsi_Cmnd` typedef	24	3508 (688)	3	1 year	0	9	390x	0.4s (1.8s)	4	83%
M11. remove ACPI tracing macros	32	9311 (2252)	3	4 months	1	41	227x	0.2s (0.5s)	1	97%
M12. use the new `IRQF_` constants	525	8750 (1548)	3	3 months	0	36	243x	0.4s (10.5s)	0	100%
M13. remove `.owner` fields	304	3909 (341)	5	2 months	1	39	100x	0.3s (5.2s)	0	100%
M14. remove `dev_link_t` and `client_handle_t`	45	9381 (2727)	1	1 day	0	661	14x	4.8s (34.8s)	4	91%
M15. replace `MODULE_PARM` by `module_param`	55	1451 (290)	5	10 months	2	113	13x	0.5s (2.9s)	16	71%
M16. reuse existing `.owner/.name` fields	128	3039 (445)	3	5 months	1	106	29x	0.3s (1.5s)	7	95%
M17. remove unnecessary casts	22	2378 (505)	2	12 days	0	14	170x	0.1s (0.2s)	0	100%
M18. replace `kcalloc(1,...)` with `kzalloc`	178	4100 (652)	5	1 year	0	16	256x	0.2s (2.2s)	1	99%
M19. convert `dvb_frontend_ops` field to a structure	55	3940 (1266)	1	1 day	1	96	41x	0.3s (1.3s)	8	85%
M20. `constify` make `ethtool_ops` structures	100	1814 (316)	1	1 day	0	16	113x	0.3s (2.0s)	2	98%
M21. changes in INIT_WORK	245	15445 (3264)	11	1 month	0	493	31x	2.6s (90.0s)	90	63%
M22. use the `platform_driver` type	97	9784 (2971)	8	1 year	17	210	47x	0.5s (6.1s)	19	80%
M23. rename `get_property`	59	2553 (313)	1	28 days	0	31	82x	0.2s (0.7s)	1	98%

Figure 7: Mega collateral evolutions

single programmer. Many driver maintainers may end up applying the collateral evolution, over a long period of time. Automation and documentation are highly desirable here, both to speed up the collateral evolution process and to aid the maintainers of drivers outside the kernel source tree who have to update their code.

To study the phenomenon of mega collateral evolutions, we have used `patchparse` to collect the set of changes that occur 100 or more times between Linux 2.6.12 and Linux 2.6.20. From these changes, we have identified 35 mega collateral evolutions, of which, due to time constraints, we have studied 23 in detail, as listed in Figure 7. In each case, we developed the semantic patch by studying the patterns occurring in only a small subset of the relevant drivers. This exemplifies the process that would typically be carried out by a library developer who has to update all affected drivers across the Linux kernel source tree.

The left side of Figure 7 describes the collateral evolution process in the manual case. The number of relevant files ranges from 22 (with many relevant code sites in each file) to over 1000. The collateral evolutions in these files lead to over 100,000 lines of patch code; the column "patch LOC" gives the total number of lines of code in a patch affecting a driver file, including patch code for header files and non-driver files, as an indication of the number of lines of code that must be scanned through by a driver maintainer to understand the collateral evolution, while the column "change LOC" gives only the number of added and removed lines in the driver C files, as an indication of the amount of work that is required. In some cases, the collateral evolutions were all done by a single person, who submitted all relevant patches within the same day (although the work was probably done over a longer period). In other cases, the collateral evolutions were done by up to over 40 people, who submitted their patches over a period of up to one or two years. While there are only 166 maintainers in all, this does not include those who have to update the drivers outside the kernel source tree. Indeed, two members of our research group were recently confronted with the task of updating a third-party driver so that it would work with the latest version of Linux. Finally, programmers made errors or missed collateral evolution sites in several cases, including in 39 files in M2 and in 25 files in M3.

The right side of Figure 7 describes the collateral evolution process with Coccinelle. Some collateral evolutions involve only a uniform renaming or removal of code while others require considering many variations and interrelated code fragments that may be scattered throughout the file. Thus, the semantic patches range in size from two lines to remove an include file (M9) up to over 600 lines to capture the specific treatment required for each of a large set of functions (M14). While Coccinelle times out for three files (in M3 and M21; we use a timeout of 90 seconds), the average execution time per `.c` file that is affected by the semantic patch is only 0.7 seconds. In general, larger semantic patches take more time, but the complexity of the code, particularly the number of nested loops in the functions affected by the collateral evolution, can also have an impact.

For 94% of the files overall, the semantic patch performs the collateral evolution correctly. In the few remaining files, the transformation engine overlooks an affected code site, typically either because of dataflow or interprocedural effects, for which support is currently limited. M15 and M21 have the lowest percentages of correctly handled files (71% and 63% respectively). For M15, in many cases it is necessary to reason about whether various top-level entities are defined before or after each other in the file. Because this information is not typically relevant to API evolutions, Coccinelle currently treats each top-level element in isolation, with no information about its position. For M21, which also has the second-largest semantic patch, there are many special cases to take into account and the rule is not yet complete. Nevertheless, in both cases, the documentation and automation provided by the semantic patch can offer a good foundation for the remaining manual transformations.

M5, in which the semantic patch updates fewer than 80%

	Manual evolution			Coccinelle				
Evolution	Files	Patch LOC (Change LOC)	Miss & Err.	SP LOC	Patch/ SP LOC	Running time Avg. (Max.)	Miss	% OK
C1. rename mem_map_(un)reserve functions	30	1401 (138)	1	20	70x	0.4s (0.7s)	0	100%
C2. reorganize i2c_client structure	20	1593 (377)	7	104	15x	0.3s (0.7s)	0	100%
C3. `proc_info`	19	2416 (218)	3	63	38x	0.5s (1.9s)	0	100%
C4. add lock in interrupt callbacks	28	851 (327)	7	111	8x	2.7s (29.6s)	8	73%
C5. change agp_(un)register_driver protocol	11	568 (91)	2	31	18x	0.1s (0.1s)	0	100%
C6. change type of argument in 3 block-related functions	6	385 (23)	1	54	7x	0.2s (0.4s)	0	100%
C7. eliminate fields in BCState structure	15	443 (146)	0	46	10x	0.3s (0.8s)	4	73%
C8. end_request(X) becomes end_request(CURRENT,X)	28	1716 (350)	2	5	343x	0.6s (6.6s)	1	96%
C9. drop CLEAR_INTR statement macro	26	673 (137)	0	34	20x	0.2s (1.0s)	0	100%
C10. rename fields in PStack structure and getters introduction	24	61892 (613)	3	48	1289x	0.3s (0.8s)	1	96%
C11. change in arguments of usb_(de)register_dev	9	470 (153)	1	43	11x	0.6s (1.5s)	5	44%
C12. reorganize fields of input_dev/gameport structures	55	1260 (544)	4	37	34x	0.1s (0.4s)	4	93%
C13. rename constant macro ATA_MAX_PRD	6	101 (12)	0	11	9x	0.1s (0.2s)	0	100%
C14. remove pci_present() calls in conditions	82	1944 (403)	0	89	22x	3.0s (90.0s)	16	76%
C15. rename and reorganize calls to scsi_set_pci_device	23	383 (59)	1	5	77x	1.0s (9.5s)	0	100%
C16. remove snd_magic_cast/kmalloc/... calls	165	12278 (2383)	3	66	186x	0.2s (1.2s)	9	95%
C17. rename and reorganize calls to atomic_dec	4	140 (26)	4	90	2x	3.7s (7.5s)	0	100%
C18. introducing pci_set_consistent_dma_mask	12	283 (44)	1	17	17x	0.3s (0.4s)	0	100%
C19. factorize code via tty_wakeup	62	4455 (1197)	16	74	60x	0.5s (1.7s)	9	86%
C20. rename pci_alloc/free_consistent functions	3	145 (40)	2	8	18x	0.2s (0.3s)	0	100%

Figure 8: Error-prone collateral evolutions

of the drivers correctly, illustrates the potential benefit of the documentation that Coccinelle provides. This collateral evolution primarily involves renaming the type `termios`, but in 23 files, the library developer additionally added initialization of some structure fields. In another 22 files, the same conditions appear to hold, but the initialization is not added. It may be that the code in some other part of the file determines whether the initialization should be introduced, but if there are no changes near this part of the code, it will not be present in the patch file. A semantic patch on the other hand, specifies all relevant context code and no other, which can aid the driver maintainer in inferring the conditions under which the collateral evolution should be performed.

5.4 Error-prone evolutions

We next consider some collateral evolutions that have proved difficult for developers, as evidenced by the errors that have been introduced. For this, we have used `patchparse`, as described above, to find collateral evolution candidates from Linux 2.5 and 2.6. We have furthermore used `grep` to check for files that were relevant to the collateral evolution but overlooked by the developers who created the patch files. From this analysis, we have identified the 20 collateral evolutions that are listed in Figure 8. As for the mega collateral evolutions, these range from simple to complex.

In Figure 8, the column "Errors" indicates the number of driver files in the (traditional) patches where at least one error occurred. Some collateral evolutions (C7, C9, C13, C14) have 0 errors as the collateral evolution was performed correctly for the files in the traditional patch, but some relevant files were overlooked. Errors are either due to mistakes or to conflicts with other modifications done concurrently on the Linux kernel by other developers. Errors that we have observed include neglecting to delete a local variable that then shadows an added parameter, adding code that uses variables that are defined at other collateral evolution sites but not the current one, neglecting to adjust some uses of a variable that changes type, deleting too much code, skipping some collateral evolution sites, and introducing syntax errors.

Overall, in 90% of the driver files, the semantic patch performs the collateral evolution correctly. For one file there is a timeout, as the number of nested loops causes state explosion in the model checker. The difficulties in the remaining cases are as for the mega collateral evolutions.

5.5 Updating a complete directory

When the maintainer of a driver that is not inside the kernel source tree would like to be able to use his driver with a new Linux version, he has to perform on his own all of the collateral evolutions that concern his driver, between its current version and the new one. To simulate this situation, we consider the problem of updating all of the drivers in a single directory from Linux 2.6.12 to the recent version 2.6.20. To select the directory, we have applied `patchparse` to each of the subdirectories of `drivers` and `sound` to obtain the number of commonly occurring changes found in each case, and taken `drivers/bluetooth`, which is at the median, as representative of the typical case.

The complete set of collateral evolutions affecting the `drivers/bluetooth` directory are those listed in Figure 9 plus four others for which we were not able to write semantic patches, typically because we were not able to fully understand the collateral evolution from the available information. Of the collateral evolutions affecting `bluetooth`, seven were previously identified as mega collateral evolutions, and all but one of the others are common to a number of other directories. Developing a library of semantic patches for this directory thus requires very little directory-specific work, and a driver maintainer can assume that the semantic patches have been tested on drivers exhibiting a wide variety of coding styles. All of the semantic patches apply correctly to all of the bluetooth files.

6. CURRENT LIMITATIONS

The experiments described in Section 5 show that Coccinelle is expressive enough to specify transformations involving a wide range of C and `cpp` constructs, and powerful enough to apply them to a wide range of driver code. Despite these good results, we have observed some limita-

	Manual evolution		Coccinelle			
Evolution	Files	Patch LOC (Change LOC)	SP LOC	Patch LOC/ SP LOC	Running time Avg. (Max.)	% OK
B1. replace `kmalloc/memset` by `kzalloc` (M3)	12	224 (46)	309	1x	1.0s (1.6s)	100%
B2. move the `pkt_type` substructure	11	769 (136)	14	55x	0.2s (0.4s)	100%
B3. adding error return value to config()	4	3046 (60)	137	22x	0.4s (0.5s)	100%
B4. drop `#include <linux/config.h>` (M9)	12	35470 (12)	3	11823x	0.1s (0.1s)	100%
B5. eliminate the `pt_regs` parameter (M2)	8	26979 (28)	407	66x	0.4s (0.6s)	100%
B6. unify event and detach handlers	4	3488 (96)	69	51x	0.3s (0.3s)	100%
B7. unify event and attach handlers	4	5201 (182)	107	49x	0.2s (0.3s)	100%
B8. remove `dev_link_t` and `client_handle_t` (M14)	4	9381 (192)	661	14x	1.6s (2.2s)	100%
B9. embed `dev_link_t` into struct `pcmcia_device`	4	3990 (107)	66	60x	0.2s (0.2s)	100%
B10. introduce skb_copy_from_linear_data{_offset}	3	2408 (8)	60	40x	0.3s (0.4s)	100%
B11. release_firmware() fixes	1	232 (1)	91	3x	0.2s (0.2s)	100%
B12. make `file_operation` structures const (M4)	1	990 (2)	48	21x	0.1s (0.1s)	100%
B13. changes in INIT_WORK (M21)	1	10705 (7)	493	22x	0.8s (0.8s)	100%
B14. annotate DECLARE_WAIT_QUEUE_HEAD	1	166 (2)	11	15x	0.1s (0.1s)	100%
B15. use bitfield instead of p_state and state	4	3341 (39)	141	24x	0.3s (0.4s)	100%
B16. add pcmcia_disable_device	4	1124 (32)	97	12x	0.2s (0.2s)	100%
B17. default suspend and resume handling	4	1664 (97)	253	7x	0.5s (0.5s)	100%
B18. remove unneeded Vcc pseudo setting	4	1170 (20)	77	15x	0.6s (0.9s)	100%
B19. remove dev_list from drivers	4	2130 (72)	32	67x	0.1s (0.1s)	100%
B20. move event handler	4	1179 (24)	22	54x	0.1s (0.1s)	100%
B21. remove .owner field from struct usb_driver (M16)	4	1520 (4)	39	39x	0.1s (0.1s)	100%
B22. gfp flags annotations	2	3414 (4)	134	25x	0.3s (0.4s)	100%
B23. Kill `skb->list`	1	1677 (8)	33	51x	0.1s (0.1s)	100%
B24. transform skb_queue_len() binary tests into skb_queue_empty()	1	830 (4)	79	11x	0.1s (0.1s)	100%
B25. remove references to pcmcia/version.h	4	684 (4)	3	228x	0.1s (0.1s)	100%
B26. convert users to tty_unregister_ldisc()	1	168 (4)	5	34x	0.1s (0.1s)	100%

Figure 9: Collateral evolutions affecting the `bluetooth` directory. Patch LOC includes only patches containing `bluetooth` files.

tions of Coccinelle that have prevented some of the semantic patches from applying completely in all cases. We now discuss the most important of these limitations, and consider how the partial matches provided by Coccinelle can aid the driver maintainer in these cases.

Currently, the Coccinelle transformation engine detects intra-procedural control-flow relationships, but not data-flow relationships or inter-procedural control-flow relationships. These relationships need to be taken into account when the programmer names a complex subterm that is relevant to the collateral evolution, or breaks a complex function into multiple helper functions. If the data-flow or interprocedural control-flow follows a regular pattern, then a transformation rule can be rewritten to explicitly take this case into account. For example, the pattern `f(g());` can be rewritten as `x=g();...f(x);`, where `x` is a metavariable. But this solution is insufficient for the general case, where an arbitrary number of aliases are introduced or a computation is broken into functions at arbitrary points. In other cases, code is not factorized into another function but into a macro. A macro can interact in arbitrary ways with its calling context, including referring to local variables and labels. In our examples, this mainly poses a problem when a semantic patch relies on type information, as such information is only available for global variables defined before the macro definition. We are working on adding more general data-flow and interprocedural control-flow relationships to Coccinelle.

A semantic patch describes code that is to be added by referring to the code to which it is to be attached. In some cases, such as the addition of an include file or a structure field initialization, there may be many valid positions for the new code, and the human programmer may choose between them based on aesthetic considerations that are difficult to encode precisely. This can lead to a proliferation of rules that consider the many possible cases and that are still not entirely successful at reproducing the strategy of the programmer. SmPL already allows a rule to specify that an include should be added at the end of a list of includes with a specific prefix, such as `linux`, which is sufficient in some cases. For structure field initializations, it may be possible to extend the language to, *e.g.*, integrate an initialization at a point that corresponds to the position of the field declaration in the structure type.

Currently, Coccinelle does not transform the contents of strings, such as those used in debugging. Indeed, strings tends to be more free form than executable code, and thus it seems difficult to write transformation rules that are generally applicable. This can, however, prevent complete application of a collateral evolution such as M3 in which a value is deleted, as it is not possible to remove references to it in any print statements in which it occurs.

Partial matches. The limitations due to data-flow effects, inter-procedural effects, and interactions with macros typically cause a semantic patch not to apply at some relevant sites. If the user believes that a collateral evolution should be applied in a certain driver, *e.g.*, because the driver uses the affected API, he can request that Coccinelle provide information about partial matches. For example, Figure 10 illustrates a simplified version of the semantic patch used for collateral evolution M3 and an extract of relevant driver code. Although the driver code involves the functions `kmalloc` and `memset` that are transformed by the semantic patch, the semantic patch does not match because the location storing the result of calling `kmalloc` is renamed and the new name is used in the call to `memset`. Detecting this case requires using data-flow information. The partial matches returned by Coccinelle indicate that a call to `kmalloc` and

The semantic patch:
```
@ r @ expression x, E1, E2; @@
- x = kmalloc(E1,E2)
+ x = kzalloc(E1,E2)
  ...
- memset(x,0,E1);
```

An extract of `drivers/mtd/onenand/onenand_bbt.c`:
```
this->bbm = kmalloc(sizeof(struct bbm_info), GFP_KERNEL);
if (!this->bbm) return -ENOMEM;
bbm = this->bbm;
memset(bbm, 0, sizeof(struct bbm_info));
```

The resulting partial match:
```
in rule: r
pattern: x = kmalloc(E1,E2) matches on line: 233
with environment: x = this->bbm, E1 = sizeof(struct bbm_info)
                  E2 = GFP_KERNEL

pattern: memset(x, 0, E1); matches on line: 239
with environment: x = bbm, E1 = sizeof(struct bbm_info)
```

Figure 10: `kmalloc/memset` partial matches

a call to `memset` were matched, the former with an environment in which the metavariable `x` is bound to `this->bbm` and the latter with an environment in which the same metavariable is bound to `bbm`. The user is thus informed that this is a potential matching site, but there is an incompatibility in the metavariable bindings. He can then either extend the semantic patch or treat the given driver manually.

7. IMPACT ON LINUX

In addition to reproducing previous collateral evolutions, we have also developed several semantic patches to complete collateral evolutions in files that were previously overlooked and to automate various tasks identified by the Kernel Janitor Project.[4] These semantic patches apply not only to device drivers, but also to files in other Linux subsystems.

We have submitted patches generated using these semantic patches to the Linux community, with the semantic patch included in the commit log. As we are not kernel developers, some of our semantic patches did not perform the right transformations in all situations. In some of these cases, developers from the Linux kernel mailing list who had no previous exposure to SmPL were able to read the semantic patch and propose appropriate corrections.

Several of the patches we have submitted have been accepted into the latest version of Linux, affecting over 150 files. Others have been accepted into a subsystem maintainer's tree, or lost in the flood of patches submitted to the kernel. Indeed, it is common for a kernel developer to have to submit a patch multiple times before it is accepted. A developer who updates his code manually may have to redo the changes each time, while when using Coccinelle, we can simply apply the semantic patches again.

8. 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. Other influences include the *Structured Search and Replace* (SSR) facility of IDEA [18], which allows specifying patterns using metavariables and provides some isomorphisms, the work of De Volder on JQuery [3], which uses Prolog logic variables for browsing source code, and the work of Lacey and de Moor on formally specifying compiler optimizations using CTL [11].

[4]http://janitor.kernelnewbies.org/

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, as originally designed, however, apply to the whole program, while in Linux, 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 [9]. 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 fixed set of Eclipse refactorings.

A number of program transformation frameworks have recently been proposed, targeting industrial-strength languages such as C and Java. CIL [19] and XTC [8] are essentially parsers that provide some support for implementing abstract syntax tree traversals. No program transformation abstractions, such as pattern matching using repeated metavariables, 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 language for writing program transformations [24]. Convenient pattern-matching is 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 and others, such as control-flow analysis, have to be implemented in the Stratego language. This leads to rules that are very complex for expressing even simple collateral evolutions.

A number of tools for bug-finding have recently been targeted toward operating systems code, including the work of Engler *et al.* [4] and Microsoft's SDV [1]. These tools generate reports of possible bugs that the driver maintainer has to check and correct by hand. Nevertheless, there is no explicit direction on how to construct the bug fix. To address this problem, Weimer has proposed to infer automatically a possible bug fix that both satisfies the verification rule that prompted the bug report and is close to the code found in the original source program [25]. While our semantic patches are directed towards transformation, not bug finding, they do show explicitly how to construct the new code. Furthermore, when a collateral evolution has already been done manually, it is possible to detect bugs by applying the semantic patch to the old code and then comparing the result to the updated code created by hand.

Analysis tools in Linux. The Linux community has recently begun using various tools to better analyze C code. Sparse [21] 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`. Sparse-detected bugs are mentioned regularly in Linux patches. Smatch [22] is a similar project developed by the Linux kernel janitors and enables a programmer to write Perl scripts to analyze C 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.

9. CONCLUSION

In this paper, we have proposed a transformation tool, Coccinelle, for documenting and automating device driver collateral evolutions. Our experiments with the Coccinelle prototype on 62 collateral evolutions, illustrating a wide range of situations, demonstrate the practicality of the approach. In most cases, the semantic patches have been successfully applied to the vast majority of the relevant files. Coccinelle has even identified code sites that were not correctly updated in the original patches. This merely emphasizes the need for improved tool support and more comprehensive documentation of collateral evolutions.

Our study shows that it is possible to provide tools that can help the OS developer manage internal OS evolution in a easier way, with more confidence. In practice, evolution in an OS is essential, if it is to keep up with the continuous evolution of hardware (new buses, new processors, etc.). In fact, this need for evolution is common to all OSes, however, the developers of Linux have chosen to face evolution most directly while others stack layers upon layers leading to an intricate software architecture. Therefore, we believe that the use of Coccinelle is not limited to Linux, and could be envisioned in any systematic evolution process.

As a next step, we plan to design semantic patches to document all collateral evolutions in recent versions of Linux. Such an extensive study will benefit both industrial and academic developers of dedicated drivers and Linux variants. We are also investigating other uses of semantic patches, as guidelines for improving code quality, as API documentation, and for bug finding.

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.

Availability

Coccinelle, the semantic patches, the default set of isomorphisms, and the driver files used in our experiments are available on our web page:
`http://www.emn.fr/x-info/coccinelle/`.

10. 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 Eurosys'06 [5], pages 73–85.

[2] 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.

[3] K. De Volder. JQuery: A generic code browser with a declarative configuration language. In *8th International Symposium on Practical Aspects of Declarative Languages*, 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 *OSDI'00*, pages 1–16, San Diego, CA, Oct. 2000.

[5] *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, Leuven, Belgium, Apr. 2006.

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

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

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

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

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

[11] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In *Compiler Construction, 10th International Conference*, LNCS 2027, pages 52–68, Genova, Italy, Apr. 2001.

[12] 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 *OSDI'04*, pages 289–302, San Fransisco, CA, Dec. 2004.

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

[14] 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`.

[15] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *USENIX Winter*, 1994.

[16] B. McCloskey and E. Brewer. Astec: a new approach to refactoring c. In *ESEC/FSE-13: 10th European software engineering conference/13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–30, New York, NY, USA, 2005. ACM Press.

[17] A. Morton. Patch management scripts, Oct. 2002. `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] 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*, LNCS 2304, pages 213–228, Grenoble,

France, Apr. 2002.

[20] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In Eurosys'06 [5], pages 59–71.

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

[22] The Kernel Janitors. Smatch, the source matcher, June 2002. `http://smatch.sourceforge.net`.

[23] L. Torvalds. Linux kernel coding style. `linux/Documentation/CodingStyle`.

[24] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, LNCS 3016, pages 216–238. Spinger-Verlag, 2004.

[25] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, Portland, Oregon, USA, Oct. 2006.

APPENDIX

In this appendix, we give the definitions of the first ten semantic patches described in Figure 8. We have reformatted and simplified most of the semantic patches to be able to give a broad overview. The complete semantic patches are available from our web site.

C1. This semantic patch essentially just replaces one set of functions by another. The first two rules illustrate how it is possible to define transformations involving preprocessor code. The third rule specifies a disjunction of possible transformations; the first one that matches is used.

```
@@ identifier page; @@
- #define cs4x_mem_map_unreserve(page) mem_map_unreserve(page)

@@ identifier page; @@
- #define cs4x_mem_map_reserve(page) mem_map_reserve(page)

@@ expression E; @@
(
- mem_map_reserve(E)
+ SetPageReserved(E)
|
- mem_map_unreserve(E)
+ ClearPageReserved(E)
|
- cs4x_mem_map_unreserve(E)
+ ClearPageReserved(E)
|
- cs4x_mem_map_reserve(E)
+ SetPageReserved(E)
)
```

C2. In this case, we show only the first rule, which moves a field from the top-level of a data structure initialization into a substructure. Because structure initializers are unordered, the initialization of the field `name` is removed from wherever it occurs. An added initialization, on the other hand, has to be specified adjacent to some syntax element, to which it will be attached. Here the initialization of the `dev` field is specified to be added at the end of the initializer.

```
@@ identifier I; expression E; @@
struct i2c_client I = {
-       .name = E,
...
+       .dev = { .name = E, },
};
```

C3. This semantic patch appears in Sections 3 and 5.2.

C4. This semantic patch consists of two parts, one to add locks in interrupt callbacks and another to make some other transformations that were made at the same time in some of the files. We show only the code for the former. `rule1` identifies the interrupt function by the structure field in which it is stored. `rule2` finds a particular test and replaces it with code to take a lock. All locking code within remainder of the function body is removed, and a unlock is added before any return. If no return is present at the end of a function, Coccinelle considers that a `return;` is implicitly present, and thus unlock is added in that case as well.

```
@ rule1 @ struct IsdnCardState cs; identifier interrupt; @@
cs.irq_func = &interrupt;

@ rule2 @
identifier intno, dev_id, regs, cs;
statement S; identifier rule1.interrupt;
@@
interrupt(int intno, void *dev_id, struct pt_regs *regs) {
  ...
  struct IsdnCardState *cs = dev_id;
  ...
- if (!cs) { ... return; }
+ spin_lock(&cs->lock);
  <...
(   // more kinds of locking are considered in
- spin_lock(...); // the actual implementation
|
- spin_unlock(...);
)
  ...>
+ spin_unlock(&cs->lock);
  return;
}
```

C5. This semantic patch introduces a new structure value. The metavariable representing the name of the structure is declared as a `fresh identifier`, and thus Coccinelle requests it interactively from the user for each match.

```
@ rule1 @
fresh identifier agp_driver_struct; identifier fn, ent, d;
@@
+ static struct agp_driver agp_driver_struct = {
+   .owner = THIS_MODULE,
+ };
fn (struct pci_dev *d, struct pci_device_id *ent) {
    ...
(
-   agp_register_driver(d);
+   agp_driver_struct.dev = d;
+   agp_register_driver(&agp_driver_struct);
|
    if (...) {
      ...
-     agp_register_driver(d);
+     agp_driver_struct.dev = d;
+     agp_register_driver(&agp_driver_struct);
      ...
      return 0;
    }
)
    ...
  }
@@ identifier rule1.agp_driver_struct; @@
- agp_unregister_driver();
+ agp_unregister_driver(&agp_driver_struct);
```

C6. This semantic patch consists of a number of rules, each considering a different way in which a value of the appropriate type may be available. In the last rule, `when` is used to skip over all of the declarations, which do not match a statement metavariable *S1*, to the first statement.

```
@@ identifier driver, minor; @@
- set_blocksize(mk_kdev(driver->channel->major, minor),
-               CD_FRAMESIZE);

@@ kdev_t x; identifier fn, bdev; expression E; @@
 fn(..., struct block_device *bdev, ...) {
 <...
(
- block_size(x)
+ block_size(bdev)
|
- set_blocksize(x, E)
+ set_blocksize(bdev, E)
)
 ...>
}

@@ kdev_t x; identifier bdev; expression E; @@
  struct block_device *bdev;
  <...
(
- block_size(x)
+ block_size(bdev)
|
- set_blocksize(x, E)
+ set_blocksize(bdev, E)
)
  ...>

@@ kdev_t dev; fresh identifier bdev; statement S,S1;
identifier bsize; identifier fn; @@
 fn(...) {
  ...
+ struct block_device *bdev = bdget(kdev_t_to_nr(dev));
- unsigned bsize = block_size(dev);
+ unsigned bsize = block_size(bdev);
  ... when != S1
+ bdput(bdev);
  S
  ...
}
```

C7. This semantic patch removed some structure field initializations, and then moves the associated values into the initializer for another structure.

```
@r1@ struct BCState *b;struct IsdnCardState *i;identifier f,g; @@
<...
(
- b->BC_SetStack = f;
|
- i->bcs->BC_SetStack = f;
|
- b->BC_Close = g;
|
- i->bcs->BC_Close = g;
)
...>

@r2@ identifier str;struct BCState *b;struct IsdnCardState *i; @@
(
i->bc_l1_ops = &str;
|
b->cs->bc_l1_ops = &str;
)

@@ identifier r2.str, r1.f, r1.g; @@
struct bc_l1_ops str = {
  ...
+ .open = f,
+ .close = g,
};

@@ struct IsdnCardState *i; @@
(
- i->bcs->BC_SetStack
+ i->bc_l1_ops->open
|
- i->bcs->BC_Close
+ i->bc_l1_ops->close
)
```

C8. This collateral evolution is quite simple. However, in the original patch it was applied both to calls to `end_request` and to calls to `swimiop_send_request`, which is completely unrelated.

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

C9. The first four rules essentially replace a macro by its definition. In the third rule a global variable declaration is inserted after the last include that refers to a non-local file. The remaining rules remove uses of `CLEAR_INTR`. The second to last rule is a special case of this removal which leaves a single statement in the branch of a conditional. In this case, the braces are removed as well.

```
@ rule0 @ expression E; @@
DEVICE_INTR = E

@ rule1 @ identifier X; @@
- #define DEVICE_INTR X

@ depends on rule0 @ identifier rule1.X; @@
#include <...>
+ static void (*X)(void) = NULL;

@@ identifier rule1.X; @@
- DEVICE_INTR
+ X

@ depends on rule0 @ identifier rule1.X; @@
-   CLEAR_INTR;
+   X = NULL;

@@ statement S; @@
  if(...)
- {
-   CLEAR_INTR;
    S
- }

@@ @@
-   CLEAR_INTR;
```

C10. In this collateral evolution a collection of structure fields are renamed. The first rule performs the transformation required when a structure field reference occurs on the left hand side of an assignment, and the second rule performs the transformation required when a structure field reference occurs in function position. There are seven affected fields. We show only one, for conciseness.

```
@@ expression E1, E2; @@
- E1->l1.l1l2 = E2;
+ E1->l2.l1l2 = E2;

@@ expression E1, X, Y; @@
- E1->l1.l1l2(E1, X, Y)
+ L1L2(E1, X, Y)
```