Towards Documenting and Automating Collateral Evolutions in Linux Device Drivers

Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, Gilles Muller

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards Documenting and Automating Collateral Evolutions in Linux Device Drivers

Yoann Padioleau — René Rydhof Hansen — Julia L. Lawall — Gilles Muller

N° 6090

Janvier 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6090--FR+ENG

INRIA
RENNES

Towards Documenting and Automating Collateral Evolutions in Linux Device Drivers

Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, Gilles Muller

Thème COM — Systèmes communicants
Projet Obasco

Rapport de recherche n° 6090 — Janvier 2007 — 18 pages

Abstract: Collateral evolutions are a pervasive problem in Linux device driver development, due to the frequent evolution of Linux driver support libraries and APIs. Such evolutions are needed when an evolution in a driver support library affects the library's interface, entailing modifications in all dependent device-specific code. Currently, collateral evolutions in Linux are done "nearly" manually. The large number of Linux drivers, however, implies that this approach is time-consuming and unreliable, leading to subtle errors when modifications are not done consistently.

In this paper, we describe the development of a language-based infrastructure, Coccinelle, with the goal of documenting and automating the kinds of collateral evolutions that occur in device driver code. Because Linux programmers are accustomed to manipulating program modifications in terms of patch files, we base our language on the patch syntax, extending patches to *semantic patches*.

We report our initial usage of Coccinelle on a range of the collateral evolutions identified in an earlier study. For many of the collateral evolutions we have considered, Coccinelle can update 70% to 100% of the relevant Linux drivers fully automatically, with the remaining drivers requiring some manual adjustments due to variations in coding style that are not yet taken into account by our tool. We have additionally identified a number of drivers where the maintainer made some mistake in performing the collateral evolution, but Coccinelle transforms the code correctly. Our approach both eases and improves the robustness of the evolution process, and can address a variety of the problems that driver maintainers face in understanding and applying collateral evolutions in practice.

Key-words: operating system, device driver, program transformation, software reengineering, software maintenance.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 —Télécopie : +33 2 99 84 71 71

Vers la documentation et l'automatisation des évolutions collatérales dans les pilotes de périphériques sous Linux

Résumé : Les évolutions collatérales sont un problème important dans le dévelopement des pilotes de périphériques sous Linux due à l'évolution fréquente des APIs et des bibliothèques de support aux pilotes sous Linux. Ces évolutions collatérales sont requises lorsqu'une évolution dans une bibliothèque de support aux pilotes affecte l'interface de cette bibliothèque, entraînant des modifications dans le code des pilotes qui dépendent de cette bibliothèque. Actuellement, ces évolutions collatérales sont faites presque manuellement. Cependant, le grand nombre de pilotes rend cette approche très coûteuse en temps et peu fiable, entraînant des erreurs subtiles lorsque les modifications ne sont pas faites de manière cohérente.

Dans ce papier, nous décrivons le dévelopement d'une infrastructure, Coccinelle, basé sur un langage avec pour but de documenter et d'automatiser le genre d'évolutions collatérales se produisant dans le code des pilotes de périphériques. Comme les programmeurs Linux sont habitués à manipuler des modifications de programmes en terme de fichiers *patch*, nous avons basé notre langage sur la syntaxe des patches, étendant les patches en des *patches sémantiques*.

Nous faisons le rapport de notre première utilisation de Coccinelle sur un ensemble d'évolutions collatérales identifiées lors d'une étude précédente. Pour beaucoup des évolutions collatérales que nous avons considérées, Coccinelle peut mettre à jour entre 70% à 100% des pilotes de périphériques pertinents de manière complètement automatique, et requiert des ajustements manuels pour le reste des pilotes due à des variations dans le style du code qui ne sont pas encore gérées par notre outil. De plus, nous avons identifié un certain nombre de pilotes où le mainteneur avait commis des erreurs lorsqu'il avait fait l'évolution collatérale manuellement, mais où Coccinelle transforme le code correctement. Notre approche à la fois facilite et améliore la robustesse du processus d'évolution du code, et peux adresser une grande variété des problèmes auxquels est confronté un programmeur lorsqu'il veut comprendre ou appliquer des évolutions collatérales.

Mots-clés : système d'exploitation, pilote de périphérique, transformation de programme, génie logiciel, maintenance de programme.

Contents

> *"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, Linux Symposium July 2006.

1 Introduction

One of the most challenging aspects of OS development is to keep up with the requirements of new devices. While commercial OSes have tended to favor maintaining backwards compatibility, the developers of Linux have chosen a more flexible development model, in which internal libraries and APIs frequently evolve to improve performance, enhance security, and meet new hardware requirements. This continual state of flux, however, induces a massive maintenance effort, to bring dependent device-specific code up to date with the evolutions in libraries and APIs. Simple examples 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, where the actual code transformation is highly dependent on the specific context in which it occurs.

In previous work, we have given these code modifications the name *collateral evolutions* [22]. Collateral evolutions are a concern to all of the actors involved in driver development and maintenance: *subsystem maintainers* who are responsible for maintaining the driver support libraries and the dependent device-specific code in the Linux kernel source tree, *device experts* who develop and maintain device-specific files outside the kernel source tree, and *motivated users* who find that their hardware is not adequately supported. The potential for mistakes, misunderstandings, and conflicts in this highly distributed effort has contributed to the continuing unreliability of driver code [2].

We have additionally previously quantified the collateral evolution problem, using ad hoc data mining tools that we have developed [22]. Driver support library evolution is increasing with time, 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. Associated with these evolutions, we have detected collateral evolutions that require modifications in up to almost 400 files, at over 1000 code sites. We have furthermore manually studied 90 library evolutions, affecting over 1600 device-specific files. Our results have shown that the sheer number of device drivers and the greatly varying expertise of device driver developers and maintainers has made collateral evolutions difficult, time-consuming, and error-prone in practice.

These issues clearly call for a formal means of describing collateral evolutions and automated assistance in applying them. Conventional wisdom has it that automatic program transformation tools cannot be successful in the context of device drivers, because they are written in C, which is difficult to analyze and admits a wide variety of coding styles. The driver code affected by collateral evolutions, however, has some properties that alleviate these difficulties. First, related interface elements are often used within a single function or at a very shallow function-call depth, restricting the amount of code that needs to be analyzed and thus making precise analyses possible. Second, the structure of code using interface functions is largely dictated by the constraints imposed by the library, and thus is mostly impervious to coding style. Finally, many drivers are written by copy-paste, where new device-specific code is created by updating existing device-specific code according to the specific properties of the new device [15]; since code that interacts with the driver support libraries is typically not device-specific, the code affected by collateral evolutions may be syntactically identical across many drivers. These properties hold out hope that an automated transformation system that is tuned to the properties of device driver collateral evolutions can help address the collateral evolution problem.

We have begun developing a language-based infrastructure, Coccinelle, with the goal of documenting and automating the kinds of collateral evolutions that are required in device driver code. Building on the requirements that we have observed in our study of Linux drivers, this infrastructure provides several novel features. First, unlike many transformation systems, which describe how to create code in terms of actions to perform, we take a more WYSIWYG approach, describing collateral evolutions as *semantic patches*, which like traditional patches describe the original code and the updated code in terms of fragments of ordinary C code. The specifications are thus easy to correlate to driver code, while providing concise but formal specifications of the collateral evolution. Second, our semantic patches are applied by considering not only the affected code's text, as in the case of traditional patches, but also its syntax and semantics. This approach abstracts away from device-specific computations and individual coding style, implying that a single semantic patch can be applied unchanged to many files. It is our aim that Coccinelle should address the needs of driver maintainers at all levels, from subsystem maintainers to device experts and motivated users, by providing a

readable but widely and automatically applicable specification of device driver collateral evolutions.

In this paper, we present initial experiments showing how Coccinelle can be used to reduce the burden of device driver collateral evolution. The main contributions are:

- An assessment of the practical problems facing driver maintainers in performing collateral evolutions.
- The design of SmPL[1], a language for specifying collateral evolutions relevant to device drivers; a SmPL specification (i.e, a semantic patch) serves both as detailed documentation of the collateral evolution and a concrete description of the needed code transformations.
- The design of the Coccinelle transformation engine for applying semantic patches to device driver code.
- A preliminary analysis of the effectiveness of the proposed language and transformation engine in addressing the problems faced by driver maintainers. We find that we can already express and carry out a wide variety of the collateral evolutions identified in our previous study, even though the current early stage prototype implementation of Coccinelle is not sufficient to express all possible transformations.
- A demonstration of the conciseness of semantic patches which for our examples are up to 343 times smaller than the total size of equivalent driver patches. This result shows that collateral evolutions can be documented in short and comprehensive specifications.

The rest of this paper is organized as follows. In Section 2, we present an overview of the issues that arise in practice in manually applying collateral evolutions to Linux drivers. We then present our approach to documenting and automating collateral evolutions: in Section 3, we present SmPL, and in Section 4, we describe the transformation engine. In Section 5, we describe some experiments in writing semantic patches for the examples illustrating the issues identified in Section 2 and applying these semantic patches to driver code. Finally, we describe some related work in Section 6 and conclude in Section 7.

2 Issues in manually performing collateral evolutions

The application of collateral evolutions in Linux has been plagued by problems of mistakes, misunderstandings, and conflicts. In this section, we describe some of the impacts that these problems can have in practice. Some of the examples used were described in more detail in our previous work [22].

2.1 Mistakes

In the drivers in the Linux source tree, mistakes seem to be the most common problem in performing collateral evolutions. Indeed, of the 90 library evolutions that we have studied in detail, 16% involved at least one typographical or inattention error in at least one device-specific file. Mistakes 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. Indeed, several such mistakes were made in applying the "proc_info" collateral evolutions that we detail in Section 3. Many mistakes cause either compile-time or link-time errors, but they are not detected by the driver maintainer performing the collateral evolution because the many compilation options of Linux make it hard to compile all code. Others cause memory errors, but these are only detected if the affected code is tested by the driver maintainer. Finally, others may cause no immediate error, as when common code blocks are abstracted into a new function but the factorization is not applied everywhere. The code at the collateral evolution site remains functional until the new function is changed in some critical way.

2.2 Misunderstandings

Misunderstandings are an issue when the person who performs the evolution in the driver support library is not the same as the person who performs the associated collateral evolutions. This typically occurs in three cases: (1) when the person performing the evolution defines a wrapper function, providing backward compatibility, rather than performing the collateral evolutions, and collateral evolutions are required when the wrapper function is subsequently removed and (2) when the collateral evolutions must be applied to code outside the Linux source tree. We provide some examples of each case.

Wrapper functions At the beginning of Linux 2.4.2, the driver initialization process changed such that use of the function `check_region` could cause a race condition, and should be replaced by the function `request_region`. Although `check_region` was now unsafe, it was redefined as a wrapper function for `request_region` and almost no drivers were initially

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

updated. Instead, over the next few years, the collateral evolution was performed by all kinds of driver maintainers, who had to work without the expertise of the developer who performed the original evolution. Discussions in Linux mailing lists show that driver maintainers and motivated users were not always sure what to do to bring their drivers up to date, as the protocol for using `request_region` is slightly different from that for using `check_region`.

Drivers outside the Linux source tree We can indirectly see evidence of the problems faced by the maintainers of drivers outside the Linux source tree from the case of drivers that have recently entered the Linux source tree.

The file `sound/oss/au1000.c`, entered Linux 2.4 after the start of Linux 2.5, and thus did not participate in the Linux 2.5 evolution process.[2] It later appeared in Linux 2.6.0. Many of the Linux 2.5 collateral evolutions were carried out in creating the Linux 2.6.0 version, but not all. One such omitted collateral evolution implied that the driver used library functions that were no longer defined. Even though this error made it impossible to build a kernel using this driver, the error was not corrected until Linux 2.6.11. A similar issue occurs in the file `sound/oss/ite8172.c`, which did evolve through Linux 2.5, but appears to be replaced by an external version at the start of Linux 2.6. In this external version, the aforementioned collateral evolution was also not done. The error was again fixed in Linux 2.6.11.

2.3 Conflicts

Conflicts may arise when a collateral evolution is performed on a version of Linux which is not the latest one, in which case the collateral evolution must be retargeted and *merged* into the latest available version. Problems during merge include (1) the introduction of new collateral evolution sites, either in drivers that already existed in the older version of Linux, or in new drivers integrated in more recent versions, that must also be evolved accordingly; (2) other perturbations to the source code that break the merge process. Such perturbations range from changing whitespace in or adjacent to the collateral evolution site to localized code changes, such as changing an argument of a function affected by a collateral evolution that itself changes the function's name.

Introduction of new collateral evolution sites On October 5, 2004 code was submitted to the Linux source tree implementing a collateral evolution that changed the types of the functions `pci_save_state` and `pci_restore_state` such that each lost their last argument. Just 15 days previously, however, code was added to the file `ne2k-pci.c` that contained calls to `pci_save_state` and `pci_restore_state`. Because the collateral evolution was based on an earlier version, these calls were not taken into account. In principle, such problems could be avoided if maintainers would always use the latest version of the Linux source tree. In practice, however, the large number of developers and threads of development makes this all but impossible.

Other changes In September 2003 the `devfs` library was officially declared obsolete and to be replaced by the udev library in the kernel. In June 2005, after an extended period of time for developers to adapt their drivers to the new library, the `devfs` library was removed and collateral evolutions removing all uses of the `devfs` library were performed in the numerous drivers that were still using it. However, only a small part of the overall change, a part that did not require change to existing code nor cause any compilation errors, was accepted into the Linux source tree. Consequently the code modifications were maintained in parallel and kept synchronised with the latest Linux version for an entire year by manually resolving any conflicts arising from changed and newly added code. Finally, in June 2006, the remainder of the modifications were accepted.

Assessment

All of these problems point to the need to document and automate collateral evolutions. Currently, patch files are the only standard support for documenting and automating changes in Linux code [17]. Patch code describes a specific change in a specific version of a single file. To create a patch, a developer must modify each source code file by hand, and then apply the automatic `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 automatic `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. There is still someone who must manually and visit and update all of the files, which remains time-consuming and error prone. Using patches as documentation to understand collateral evolutions is often done, but is not easy or reliable, because the evidence in the patch files is determined by the specific code in the patched files and not the concepts of the collateral evolutions. To address these issues, an approach is needed that describes collateral evolutions simply, so that they can easily be understood by all the actors involved, and generically, so that they can

[2] Traditionally, even numbered version of Linux have been stable versions, in which most changes are bug fixes, while odd numbered versions have been unstable version, in which there are many evolutions and collateral evolutions.

```
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`

be applied automatically to many files, both inside the Linux source tree and out.

3 SmPL in a Nutshell

To address the issues described above in performing collateral evolutions, we propose a language based approach for formally specifying collateral evolutions. In this section, we first describe a moderately complex collateral evolution that was considered in our previous study [22] and then introduce the language SmPL (Semantic Patch Language) in terms of this example.

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. As this could result in corruption of the reference count, these functions were removed from the SCSI interface [16]. 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. Note that this collateral evolution involves both functions that are defined by the library and evolve in some way, and functions that are defined by the device-specific code and must respect a prototype that is itself defined by the library and evolves in some way.

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 such calls: 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 the SmPL semantic patch describing these collateral evolutions. Overall, the semantic patch has the form of a traditional patch, 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

```
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

the case of a semantic patch, the context information declares a set of *metavariables*, not a set of line numbers as does a patch. A metavariable can match any term of the kind specified in its declaration (identifier, expression, integer 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.[3] Note that when a function definition is transformed, the corresponding prototype is also transformed automatically in the same way; 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.

[3]It is also possible, via a slightly more complex semantic patch, to identify the function to transform in terms of how it is communicated to the SCSI library, which eliminates ambiguity in case of multiple functions with the same prototype [21].

Overall, the rule applies independent of spacing, line breaks, and the presence of 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. 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 15 of Figure 2 matches a conditional that tests the value of `hostptr` using any of the listed variants. Currently, the default set of isomorphisms contains 33 equivalences commonly found in driver code.

The semantics of sequences We have noted that the SmPL construct "..." matches any sequence of instructions. In general, one may consider a sequence as a list of the instructions that are explicitly contiguous in the source code (syntax), or as a list of the instructions that are executed contiguously at runtime (semantics). In the case of SmPL, we use an approximation of the latter: the construct "..." matches any sequence of instructions in the program's control-flow graph. Note, however, that Coccinelle does not take into account runtime values in computing such paths, so the set of paths considered is an over-approximation of runtime behavior.

The strategy of matching the construct "..." against sequences in the control-flow graph rather than sequences in the source program syntax is essential for specifying device driver collateral evolutions in a generic way that is applicable to many drivers. 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, driver functions are typically structured as a tree, with many exit points. An example is the code in Figure 1a, whose control-flow graph is shown in Figure 3. As the structure of this tree depends on device-specific properties, it cannot be explicitly represented in a semantic patch. Instead, a semantic patch must express the pattern of operations required by the library, which can be described generically because it is always the same. This is achieved by using control-flow paths.

We may concretely see the utility of this approach when evolving the scsiglue proc_info function of Figure 1 which contains two calls to `scsi_host_put`, while the semantic patch of Figure 2 contains only one. The control-flow graph 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. 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 the semantic patch.

Other features SmPL contains a number of other features for matching other kinds of code patterns. Among these include the ability to match and transform a term wherever and however often it occurs analogous to the `/g` modifier of `sed`, the ability to describe a disjunction of possible patterns to be tried in order, 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 allowed uses of the interface elements or due to programmer sloppiness.

4 The Coccinelle Transformation Engine

SmPL is a language for specifying semantic patches. To apply them, we need a transformation engine that takes as input a semantic patch and a collection of drivers, identifies those drivers that are affected by the collateral evolution, and transforms them according to the semantic patch. In this section, we describe our main design decisions for this transformation engine, some of the highlights of its implementation, and our vision of how the SmPL language and the transformation engine will be used together in practice.

Design decisions 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 [3], with some additional features). The matching of the formula against the model is then implemented using a variant of a standard model checking algorithm [13]. This approach, which was inspired by the work of Lacey and de Moor on a related but simpler transformation problem [14], 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 some features that we have found useful to express the SmPL semantics and some optimizations specific to the treatment of C code.

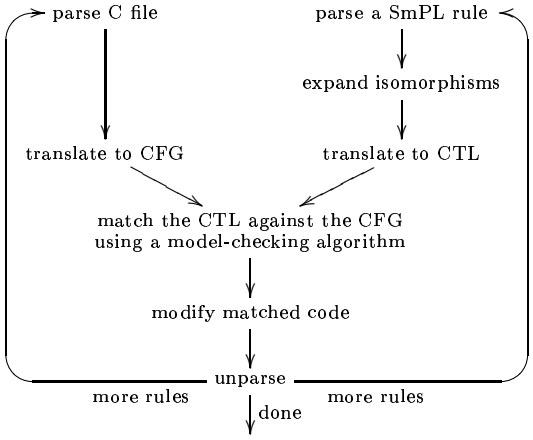

Figure 4: The Coccinelle engine

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 The main challenge in parsing the C source file is to collect enough information to be able to generate code that is readable and in the style of the original source code. As collateral evolutions are just one step in the ongoing maintenance of a Linux device driver, these features are essential 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 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. As has been found by others [10], parsing C code while handling and maintaining preprocessing directives such as `#ifdef` and `#define` and macro uses poses a significant challenge. The Coccinelle C-code parser does not expand any preprocessing directives, and instead treats them as comments. Because unexpanded macros may 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 the macro `list_for_each`, which expands into a loop header, as the start of a loop.

While not perfect, we expect these heuristics to cover the majority of the requirements of driver code. For macros, we plan to extend the tool to take into account

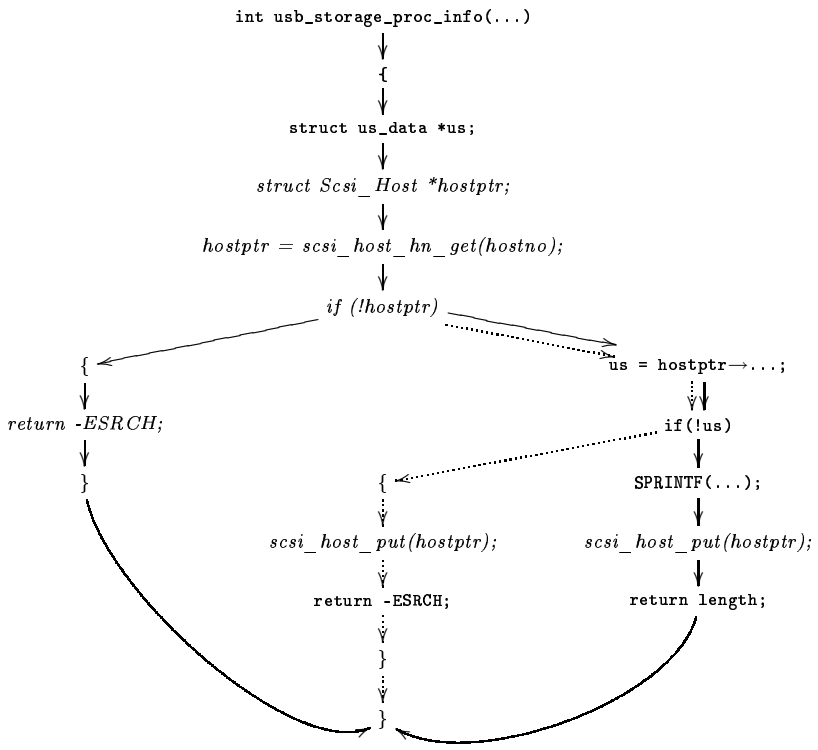

Figure 3: Control-flow graph for Figure 1 (a)

both the unexpanded and expanded code, to detect collateral evolution sites in both.

Parsing the semantic patch The main challenge 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. The arbitrary intermingling of - and + code, however, implies that the semantic patch does not satisfy the C grammar. For example, the transformation rule may have the form of a function definition that has two function headers, one to be removed and another to be added. The SmPL parser thus parses the code in two steps, first parsing the *minus slice*, consisting of the - and context code, which represents the code to match, and then the *plus slice*, consisting of the + and context code, which represent the code to generate. Once parsed, the two slices are merged such that the + tokens of the plus slice are attached to the context and - tokens adjacent to which they should be inserted. The minus and plus slices of the semantic patch of Figure 2 are shown in Figures 5a and 5b, and the merged code in Figure 5c. Note that the - and + modifiers, which were originally associated with complete lines, are now attached to individual tokens and that contiguous "..."s in the plus slice are elided.

Once the semantic patch has been parsed, the isomorphisms are applied to the resulting 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.

The final step is to translate the merged AST into our variant of CTL. Figure 6 shows a slightly simplified CTL representation of our proc_info example of Section 3 (the null test on line 13 of Figure 2 and some other details are omitted for conciseness). While the CTL code is complicated, it serves only as an assembly language, which driver maintainers do not have to read or understand. In our example, the semantic patch consists essentially 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 (AX), g is eventually found U, and on each path from f to g there is no occurrence of either f or g ($\neg(f \vee g)$). An existentially quantified variable v marks terms for which we want to record the matching nodes in the CFG, for subsequent transformation. The use of the predicate *Paren* ensures that the matched braces are corresponding open and

$$
\begin{array}{l}
\exists \mathit{hostno},\ \mathit{hostptr}\ . \\
(\exists \mathit{proc\_info\_func},\ \mathit{buffer},\ \mathit{start},\ \mathit{offset},\ \mathit{length},\ \mathit{inout},\ v\ . \\
\quad \mathit{proc\_info\_func}(\texttt{char *}\mathit{buffer}, \texttt{ char **}\mathit{start}, \texttt{ off\_t } \mathit{offset}, \texttt{ int } \mathit{length}, \texttt{ int } \mathit{hostno}, \texttt{ int } \mathit{inout})_v) \\
\quad \wedge \\
\quad \mathsf{AX}(\exists p\ . \\
\qquad (\{ \wedge \mathsf{Paren}(p) \wedge \\
\qquad \mathsf{AX}\,\mathsf{A}[\neg(\texttt{struct Scsi\_Host *}\mathit{hostptr}\texttt{;} \vee (\{ \wedge \mathsf{Paren}(p))) \ \mathsf{U} \\
\qquad\quad (\exists v\ .\ \texttt{struct Scsi\_Host *}\mathit{hostptr}\texttt{;}_v \wedge \\
\qquad\quad \mathsf{AX}\,\mathsf{A}[\neg(\mathit{hostptr}\texttt{ = scsi\_host\_hn\_get(}\mathit{hostno}\texttt{);} \vee \texttt{struct Scsi\_Host *}\mathit{hostptr}\texttt{;}) \ \mathsf{U} \\
\qquad\quad\quad (\exists v\ .\ \mathit{hostptr}\texttt{ = scsi\_host\_hn\_get(}\mathit{hostno}\texttt{);}_v \wedge \\
\qquad\quad\quad \mathsf{AX}\,\mathsf{A}[\neg(\texttt{scsi\_host\_put(}\mathit{hostptr}\texttt{);} \vee \mathit{hostptr}\texttt{ = scsi\_host\_hn\_get(}\mathit{hostno}\texttt{);}) \ \mathsf{U} \\
\qquad\quad\quad\quad (\exists v\ .\ \texttt{scsi\_host\_put(}\mathit{hostptr}\texttt{);}_v \wedge \\
\qquad\quad\quad\quad \mathsf{AX}\,\mathsf{A}[\neg((\} \wedge \mathsf{Paren}(p)) \vee \texttt{scsi\_host\_put(}\mathit{hostptr}\texttt{);}) \ \mathsf{U} \ (\} \wedge \mathsf{Paren}(p))]]))]))]))))
\end{array}
$$

Figure 6: CTL counterpart of the semantic patch of Figure 2

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

(a) Minus slice

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

(b) Plus slice

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

(c) Merged slices

Figure 5: The result of parsing the semantic patch of Figure 2

close 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 this 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. The + code may contain metavariables which have been bound to source code terms in the CTL formula against the control-flow graph. In the generated code, these metavariables are replaced by these values.

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.

Using Coccinelle in practice We envision that Coccinelle will be used as follows. Initially, a driver subsystem maintainer who modifies a library writes a *semantic patch* that describes the entailed collateral evolutions, based on his experience with the device-specific code in the kernel source tree. To test his

intuitions and bring the affected drivers up to date, he applies the semantic patch across the kernel source tree. This is an iterative process, as he identifies cases that are not taken into account by the semantic patch, refines the semantic patch accordingly, and applies it across the kernel source tree again. When he is confident that the semantic patch addresses the common cases, he publishes it to be used by the maintainers of device-specific code that is outside the kernel source tree. The semantic patch can then be read and applied by maintainers and motivated users, alike. At present, Coccinelle is not sophisticated enough to handle all possible coding styles, and thus both during the semantic patch development process and when applying the completed semantic patch to code outside the kernel source tree, it may fail to transform some cases. However, the Coccinelle engine detects contexts in which the semantic patch only partially matches the given code, and reports to the maintainer that some modification, along the lines of what is specified in the semantic patch, may be needed in these cases.

5 Experiments

In this section, we consider the use of Coccinelle in the context of the examples of Section 2. Except as noted, the driver source files come from the Linux `git` repositories `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`. Those repositories make publicly available the state of Linux before and after each commit. In our case, we extract the state of the affected drivers just before the patch performing the collateral evolution was accepted into the Linux source tree. In the remaining cases, we have used the files from a given version available at the `kernel.org` website. In all cases, the Coccinelle transformation engine was run on a Pentium 4 at 3.2GHz with 512Mb of RAM.

5.1 Collateral evolutions in which mistakes occurred

Mistakes were made in performing several of the collateral evolutions considered in Section 3. The proc_info collateral evolution is a typical example that we will discuss in more detail here.

As compared to the simplified semantic patch of Figure 2, some extensions are required to complete the implementation of the proc_info collateral evolution. The protocol for using the `scsi_host_hn_get` and `scsi_host_put` functions requires that the result of calling `scsi_host_hn_get` be tested and `scsi_host_put` be called to signal the end of use of the `hostptr` resource. This protocol is not always followed in driver code. Especially in older code, error checking is not always performed, and the evolution itself was motivated by the problem of missing calls to `scsi_host_put`. The complete semantic patch thus indicates that these two operations are optional, although they must be removed if present. We also extend the semantic patch with some other minor transformations that were performed as part of the same collateral evolution. The resulting semantic patch is 44 lines of code. Three of the default isomorphisms apply to this semantic patch.

Figure 7 lists the files affected by the proc_info collateral evolution, the number 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. 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 treat the proc_info functions, as other functions are immediately rejected by the transformation rule. Transformation of the proc_info functions completes in a few seconds for the smallest functions and less than 10 seconds for the largest functions.

5.2 Collateral evolutions in which misunderstandings occurred

We illustrate how semantic patches can address the problems of misunderstandings in performing collateral evolutions with (1) the elimination of `check_region` to illustrate the case of wrapper functions and (2) the replacement of `mem_map_reserve` and `mem_map_unreserve` to illustrate the case of files that evolve outside the Linux source tree.

`check_region` elimination Replacing `check_region` by `request_region` essentially entails moving the call to `request_region` up to replace the call to `check_region` and adding `release_region` before any intermediate return indicating an error condition. This transformation is expressed by the semantic patch shown in Figure 8. As compared to the proc_info semantic patch, this semantic patch contains a *nest*, delimited by `<...` and `...>`, which matches any number of occurrences of `return` between the calls to `check_region` and `request_region`. This nest allows the semantic patch to insert a call to `release_region` before any premature return from the enclosing function.

	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 7: Experiments with the proc_info semantic patch

While this semantic patch represents the essence of the collateral evolution, the driver code in practice exhibits a large number of variations. Common variations include the case where `check_region` is called iteratively in a loop in which case a call to `release_region` must be inserted in each control-flow path that goes around the loop, the case where the result of the call to `check_region` is connected to the test by arbitrary dataflow (*e.g.*, storing the result in a variable that is later tested, or control-flow paths that depend on variable values), and the case where `check_region` and `request_region` are used in different functions and linked by interprocedural control flow. To match the case of a loop, we simply write a second rule in the spirit of that of Figure 8 that detects the loop condition. In the current state of the prototype, Coccinelle does not automatically handle either dataflow or interprocedural control flow. For the former, most instances can be handled by adding an extra pattern that matches the case where the result of the call to `check_region` is stored in a variable and that variable is later tested. For the latter, it is possible to explicitly encode a fixed depth of interprocedural control-flow in the semantic patch, however, we have found in practice that this quickly becomes unwieldy. We will work on adding a more transparent consideration of dataflow and interprocedural control flow in the near future.

The complete semantic patch is 94 lines of code, including the loop and dataflow patterns. We have applied this semantic patch to the drivers that use `check_region` in the `scsi` and `cdrom` directories. In the former, our semantic patch is sufficient to treat 4 of the relevant 31 files and in the latter case our semantic patch is sufficient to treat 2 of the 11 relevant files. We expect that treatment of interprocedural control flow will improve these results significantly. When a semantic patch only partially matches a driver, Coccinelle informs the driver maintainer. He can then use

```
@@
expression rr1, rr2, rr3;
@@

-   if (check_region(rr1, rr2) != 0)
+   if (!request_region(rr1, rr2, rr3))
       { ... return ...; }
    <...
+   release_region(rr1, rr2);
    return ...;
    ...>
-   request_region(rr1, rr2, rr3);
```

Figure 8: Semantic patch for updating calls to `check_region`

the semantic patch as a guide in manually applying the collateral evolution to the given code.

Replacement of `mem_map_reserve` and `mem_map_unreserve` In Section 2, we noted that when the files `sound/oss/au1000.c` and `sound/oss/ite8172.c` arrived in the Linux source tree in Linux version 2.6.0, they contained uses of the API functions `mem_map_reserve` and `mem_map_unreserve` that were removed in Linux 2.5.69. The collateral evolution in this case is just a simple renaming of these functions from their old to new versions, and the removal of an include declaration. The corresponding semantic patch is shown in Figure 9. In the semantic patch the symbols `(`, `|`, and `)` in the first column express a disjunction of patterns. Matching of disjuncts is tried in the order the patterns appear, until one of the patterns matches.

The collateral evolution affects 28 files in Linux 2.5.69, each containing on average around 2800 lines of code. When the collateral evolution was performed by hand, 27 of the files were updated correctly. In the remaining file, the function `cs4x_mem_map_unreserve`, which was a local wrapper for `mem_map_unreserve`, was replaced by `cs4x_ClearPageReserved`, rather

```
@@
@@
- #include <wrapper.h>

@@
@@
- #include "drivers/sound/cs4281/cs4281_wrapper.h"

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

Figure 9: Semantic patch for updating calls to `mem_map_reserve`, etc.

than `ClearPageReserved`, as required. We conjecture that the transformation was carried out using an editor search-and-replace command that was insensitive to identifier boundaries. Coccinelle is aware of identifier boundaries, and thus updates all files correctly, in an average of 0.6 seconds each. Coccinelle additionally correctly updates the versions of `au1000.c` and `ite8172.c` that appear in Linux 2.6.0.

The collateral evolution is quite simple in this case, but the semantic patch still provides a useful record of what API functions have been deleted and what new functions should take their place, and ensures that the transformation is carried out correctly.

5.3 Collateral evolutions in which conflicts occurred

The use of semantic patches for alleviating or solving problems arising from conflicts in collateral evolutions is illustrated by (1) the removal of `devfs` and (2) the elimination of the last argument of both the `pci_save_state` and `pci_restore_state` functions.

`devfs` removal Removing `devfs` client code involves removing all calls to 6 specific functions, removing the code that uses a specific field name, renaming a constant, and finally removing an include declaration. An excerpt of the corresponding semantic patch is shown in Figure 10. The full semantic patch is 120 lines long. Previously, we have seen the use of "..." to represent an arbitrary sequence of statements. Here, they are used to represent an arbitrary expression, *e.g.* in the loop header.

The `devfs` library was used in many different Linux subsystems with many files in `drivers/` and some additional files in `arch/` and `fs/`. Running Coccinelle on the whole kernel takes 10 minutes and correctly updates 88 of the 134 relevant driver files.

As noted in Section 2, the full removal of the `devfs` client code was not immediately accepted in the Linux source tree. As a consequence, the maintainer regularly had to manually resolve the conflicts arising from changed and newly added code. For each version, an updated patch file was then re-submitted to the Linux source tree.

An example of such a conflict was the renaming of a variable that was involved in the collateral evolution. The local variable `hvc_driver` was renamed to `drv` which led to the modification of numerous lines of code including: `hvc_driver->devfs_name = "hvc"`. Consequently the patch file had to be updated to reflect that the variable had been renamed, as it could not have been applied to the updated version of the kernel with the renamed variable. In contrast, semantic patches abstract away from such details of the device-specific code. For example, in the semantic patch of Figure 10, the assignment of the `devfs_name` field represents the affected structure using the metavariable `E`, and thus the semantic patch can still be applied, in unmodified form, to the kernel version with the renamed variable.

Argument elimination Eliminating the argument of the two target functions is a relatively simple transformation described by two straightforward SmPL rules, as shown in Figure 11, totalling 10 lines of SmPL code. The semantic patch correctly updates 27 of the 37 relevant driver files with an average file size around 2600 lines and taking on average 4.50 seconds per file. It is particularly interesting to note that the semantic patch correctly updates the file `ne2k-pci.c` that was not taken into account in the original patch intended to update all calls to the functions `pci_save_state` and `pci_restore_state`. This illustrates some of the strength and robustness of semantic patches in situations where conflicts occur in collateral evolutions. The remaining ten files contain features, such as complicated macros or conditional compilation, that are not yet supported by the prototype implementation.

5.4 Other experiments

All in all, we have implemented semantic patches for 17 of the collateral evolutions derived from of the 90 library evolutions considered in detail in our previous study [22]. Figure 12 summarizes the results. Note that some information is not available for `check_region`, as there does not exist a single patch with all of the manual collateral evolutions in this case.

Overall, these results show that semantic patches are concise, particularly as compared to standard patch files, and can be applied efficiently, with a typical cost

```
@@ @@
- #include <linux/devfs_fs_kernel.h>

@@ @@
- TTY_DRIVER_NO_DEVFS
+ TTY_DRIVER_DYNAMIC_DEV

@@
identifier i;
@@
- for (i = ...; i < ...; i++)
(
- { devfs_mk_cdev(...); }
|
- { devfs_mk_bdev(...); }
|
- { devfs_mkdir(...); }
|
- { devfs_mk_dir(...); }
|
- { devfs_mk_symlink(...); }
|
- { devfs_remove(...); }
)

@@
identifier x;
identifier fn;
@@
- fn(x->devfs_name, ...);

@@
expression E;
expression E2;
@@

(
- E->devfs_name = E2;
|
- E->devfs_name[...] = E2;
)

@@
expression X;
@@
- X = devfs_register_tape(...);

@@ @@
- devfs_unregister_tape(X);
```

Figure 10: Semantic patch for removing `devfs`

```
@@
expression E1, E2;
@@

- pci_save_state(E1,E2)
+ pci_save_state(E1)

@@
expression E1, E2;
@@

- pci_restore_state(E1,E2)
+ pci_restore_state(E1)
```

Figure 11: Semantic patch for argument elimination

per file of 2 seconds or less. For most of the examples, over 80% of the affected drivers are updated correctly and in many cases all are. In the most of the remaining cases, some collateral evolutions sites are omitted, because of limitations in our current transformation engine, *e.g.* the lack of interprocedural analysis. Still, in these cases, Coccinelle warns the maintainer of partial matches. Finally, by comparing the manually updated code with the code updated using the semantic patch we have found a number of bugs in the manually updated code.

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 [18], 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 [19], which allows specifying patterns using metavariables and provides some isomorphisms, and the work of De Volder on JQuery [5], 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 fixed set of refactorings provided by Eclipse. We have found that many collateral evolutions are specific to the OS API, and thus cannot be described as part of a generic refactoring.

JunGL is a scripting language that allows programmers to implement new refactorings [25]. This lan-

	Affected Files	Avg File size LOC	SP size LOC	P size / SP size	Max time	Avg time	% Correct	Bugs detected
proc_info	19	2744	41	58	9.3s	2.7s	78.9%	3
check_region	42	3182	94	–	11.7s	2.2s	14%	–
mem_map_reserve	30	2705	16	87	1.3s	0.6s	96.6%	1
devfs	134	1711	120	111	11.9s	1.7s	65.2%	–
pci_save_state	37	2660	10	127	2.8s	0.7s	73.0%	0
sched_b_events	16	1006	60	14	2.7s	1.4s	87.5%	0
sched_d_events	9	1167	54	14	2.9s	2.1s	100%	0
atomic_dec	4	1431	85	1.6	2.3s	1.5s	100%	4
tty_wakeup	62	2085	74	60	5.7s	1.6s	83.9%	16
acpi_hw_low_level_read	6	637	10	85	0.2s	0.1s	100%	0
tqueue	18	1014	64	10	6.3s	2.2s	100%	0
end_request	28	1618	5	343	0.7s	0.2s	50.0%	3
CLEAR_INTR	26	1475	6	112	0.4s	0.2s	65.4%	0
current_valid	4	1989	9	46	0.5s	0.3s	100%	0
usb_inc_dev_use	10	2214	10	60	0.5s	0.3s	50%	0
pnp_activate_dev	24	1490	5	68	2.3s	0.3s	91.7%	2
LockPage	61	942	25	107	0.9s	0.2s	73.8%	0

Figure 12: Other experiments. SP is the semantic patch and P is the corresponding patch file obtained from the git repository

guage should be able to express collateral evolutions. Nevertheless, a JunGL transformation rule does not follow the structure of the source terms, and thus does not make visually apparent the relationship between 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.

A number of program transformation frameworks have recently been proposed, targeting industrial-strength languages such as C and Java. CIL [20] and XTC [11] 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 domain-specific language for writing program transformations [26]. Convenient pattern-matching and rule management strategies are built in, implying that the programmer can specify what transformations should occur without cluttering the code with the implementation of transformation mechanisms. Nevertheless, only a few program analyses are provided. Any other analyses that are required, such as control-flow analysis, have to be implemented in the Stratego language. In our experience, this leads to rules that are very complex for expressing even simple collateral evolutions.

Coady et al. have used Aspect-Oriented Programming (AOP) to extend OS code with new features [4, 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.

Fähndrich et al. have proposed "Compile-Time Reflection" as a means of matching over program structures and generating code from the matched information [7]. The approach is targeted towards generating new module members rather than fine-grained code transformation. Matching is restricted to matching over declarations, such as class and field declarations, rather than arbitrary code, as in our case. They do provide facilitates for collecting sets of information and generating code that explicitly manipulates these sets, which can be awkward to implement in our purely declarative framework. Nevertheless, we have not yet seen the need for this functionality in device-driver collateral evolutions.

A number of tools for bug-finding have recently been targeted toward operating systems code, including the work of Engler *et al.* [6] 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 [27]. 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 [23] 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 [24] 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. [6] 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 language-based framework, Coccinelle, for documenting and automating the collateral evolutions in device driver. To this end, we have designed SmPL, a declarative language, 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.

Our first experiments with the Coccinelle prototype have shown very promising results. We have been able to implement concise semantic patches for 17 collateral evolutions, illustrating problems of mistakes, misunderstanding, and conflicts. Each of these collateral evolutions affects, on average, more than 20 files. In most cases, the running time of Coccinelle is less than a second per file. With few exceptions 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 patch(es). This merely emphasizes the need for improved tool support and more comprehensive documentation of collateral evolutions.

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

The source code for the semantic patches and the driver files used in our experiments is available at the following URL:

`http://www.emn.fr/x-info/coccinelle/`

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] M. Fähndrich, M. Carbin, and J. Larus. Reflective program generation with patterns. In *Proc. of Generative Programming and Component Engineering, GPCE'06*, Portland, Oregon, USA, Oct. 2006.

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

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

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

[17] 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`.

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

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

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

[21] Y. Padioleau, J. L. Lawall, and G. Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. In *International ERCIM Workshop on Software Evolution (2006)*, Lille, France, Apr. 2006.

[22] 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.

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

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

[25] 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.

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

[27] W. Weimer. Patches as Better Bug Reports. In *Proc. of Generative Programming and Component Engineering, GPCE'06*, Portland, Oregon, USA, Oct. 2006.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399