Semantic Patches Considered Helpful

Gilles Muller, Yoann Padioleau
Ecole des Mines de Nantes
INRIA, LINA
44307 Nantes cedex 3, France
{Gilles.Muller,Yoann.Padioleau}@emn.fr

Julia L. Lawall, René Rydhof Hansen
DIKU
University of Copenhagen
2100 Copenhagen Ø, Denmark
{julia,rrhansen}@diku.dk

Introduction

Modern software development is characterized by the use of libraries and interfaces. This software architecture carries down even to the operating system level. Linux, for example, is organized as a small kernel, complemented with libraries providing generic functionalities for use in implementing network access, file management, access to physical devices, etc. Much of the Linux source code then consists of service-specific files that use these libraries. These libraries are also used by the many OS-level services that are maintained outside of the Linux source tree.

Reliance on libraries, however, is a double-edged sword. While libraries make it easy to create new software, any change in a library interface can break all dependent code. In the case of Linux, the libraries are currently undergoing rapid evolution, even in the so-called stable version Linux 2.6 [7]. Service-specific code must thus frequently be correspondingly updated to respect the new interfaces. We refer to these modifications as *collateral evolutions*. In the rest of this paper, we summarize the issues involved in this kind of evolution in the context of Linux, and propose a language-based approach to document and automate the required code modifications.

The collateral evolution problem

We have recently begun to study the collateral evolution problem in the context of Linux device drivers [10]. In the Linux 2.6.13 source tree, there are over 150 driver support libraries, each dedicated to a given bus or device type, and almost 2000 device-specific files, each containing code for interacting with a specific device. A device-specific file can use up to 59 different library functions from up to 7 different libraries. In all, we have found that driver code makes up around 50% of the Linux 2.6.13 source code. The number of collateral evolutions that affect these files is massive. Using an automatic tool that we have developed, we have identified over 1200 probable evolutions in Linux driver interfaces in the Linux 2.6 versions up to Linux 2.6.13, as compared to under 300 in all of Linux 2.2. We have found that an evolution can trigger collateral evolutions in up to almost 400 files, at over 1000 code sites. Among the probable evolutions we have identified, we have studied 90 in detail. These affect over 1600 device-specific files in Linux 2.5 and Linux 2.6. The collateral evolutions identified range from simply changing the name of a function, to complex transformations that involve sophisticated analysis of the usage context of each affected interface element, to construct new argument values, update error checking and error handling code, eliminate newly redundant computations, etc.

Currently, collateral evolutions in Linux are mostly done manually using a text editor, with the help of tools such as grep. The large number of Linux drivers, and complexity of the collateral evolutions, however, implies that these approaches are time-consuming and unreliable, leading to subtle errors when modifications are not done consistently. For example, an evolution in an interface may imply that a function that was formerly expected to return 0 or -1 to indicate failure is now expected to return a more informative error code, such as `-ENODEV` or `-ENOMEM`. Because the constants 0 and -1 can be used with other meanings in device-specific code, simply replacing all occurrences of 0 or -1 by one of these values is not likely to be correct. Furthermore, it is often necessary to choose between a set of possible error codes, which requires understanding the condition that caused the failure. Other collateral evolutions require multiple kinds of changes in a single file, such as changes to function prototypes, function definitions, and structure-field initializations. These scattered modifications must be kept consistent and harmonious with the existing coding style.

As Linux is an open-source operating system and involves multiple expertises, there is the further problem of the variety of actors involved: library developers, maintainers of service-specific files, and motivated users. The library developer is often a core member of the Linux development community and is highly knowledgeable about the OS. He makes changes in his library, and when these changes affect the interface, he typically updates all of the affected service-specific files that are part of the Linux source tree. His task is tedious and error prone, but he is well placed to understand both the reasons for the collateral evolutions and the structure of the affected code. Files that are overlooked by the library developer or that are outside the Linux source tree, on the other hand, must be updated by someone else. In these cases, either the collateral evolution is performed by the maintainer of the service-specific file or, in the case of orphaned code, the collateral evolution is performed by a motivated user. The maintainer is typically less familiar with the OS code than the library developer but knows his own code, while the motivated user may not be familiar with either of them. In both cases, there is a problem of transfer of expertise. The maintainer or user may not be aware that collateral evolutions are needed, and when made aware, may not understand the informal terminology used in library comments or mailing lists that describes how the collateral evolutions should be carried out. Further compounding the problem of collateral evolutions is the number of libraries that may be used by a single service-specific file. A maintainer or motivated user must keep up to date with evolutions occurring in many different sources.

Finally, our study shows that collateral evaluations are error-prone in practice. In 32% of the 90 evolutions we have studied in detail, some collateral evolution was omitted or done incorrectly. These introduced errors often persisted over many versions of Linux and some have not, as of this writing, been corrected.

Our vision

Our results clearly call for some form of automated support for collateral evolutions, both because of the difficulty and tedium of updating so many files and because of the errors that have been introduced. We envision a language-based approach, where a library developer who makes a change that affects the library interface creates a precise and readable specification of the collateral evolutions entailed by this change. The library developer then disseminates this specification to the maintainers of device-specific code, who can read the specification to understand the collateral evolutions that must be made to their files. Furthermore, we envision a tool to carry out the collateral evolutions described by the specification, that can be used by the maintainers of device-specific code. Such a tool should respect the coding style and preserve the readability of the existing code.

It is instructive to compare our vision to the current strategy for transmitting changes in Linux code, the patch file. Patch code describes a specific change in a specific version of a single file. Thus, a library developer who makes a change in an interface must modify each of the affected files by hand, and create a patch describing each modification. When a new version appears before the change is accepted for inclusion in the kernel source code, this work has to be repeated on the new code. Once accepted, patches are then distributed to users who use them to replicate the developer's modifications in their own files. This approach requires time-consuming and error-prone manual modification initially, and then produces an artifact that is only applicable to the files that are known to the library developer; no indication is provided as to how to map the collateral evolutions to other files. Our goal is essentially to unify these specific patches into a single generic version, that concisely captures the essence of the required collateral evolutions and is applicable to all relevant files. Such a specification must necessarily be described in terms of the semantics of the old and new code, rather than the details of its specific syntax. For this reason, we refer to our specifications as *semantic patches*.

The Coccinelle tool

To this end, we have begun designing a framework, Coccinelle, that includes a language, SmPL,[1] in which to express *semantic patches* that describe collateral evolutions, and a transformation tool for applying semantic patches to device-specific code. To fit with the habits of Linux programmer and to provide a legible syntax, SmPL is based on the standard patch format. While an ordinary patch only applies to specific files at specific lines, a SmPL semantic patch is mapped by the transformation tool onto the device-specific code not in terms of the syntax of the device-specific code but in terms of its semantics. A semantic patch specifies control and dataflow relationships, and thus abstracts away from *e.g.* the many ways of writing loops, the presence of unanticipated conditionals, and the use of local variables to rename intermediate values. SmPL also incorporates a collection of isomorphisms, which will ultimately be user configurable, to abstract away from *e.g.*, the many ways of testing for `NULL` and the use or non-use of macros. So far, we have used the language to write semantic patches for about two thirds of the 90 evolutions we have studied in detail. Some work,

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

however, remains on the design of the language, most notably how to express distinct paths of computations, such as those that end in an error case, in a way that is harmonious with the essentially linear patch-like syntax.

Related approaches

Designing a program transformation system targeting real-world operating systems code is a challenging task. Over the last few decades numerous program transformation systems have been developed (*e.g.*, [1, 2, 3, 11]). Many, however, have met with only limited success, either because they have tackled a too general problem, and thus have ended up being unwieldy, or because they have tackled a too specific problem, and thus have ended up being inapplicable to most real-world code. Coccinelle is targeted toward a specific problem, but one that affects a wide and well-identified code base. Furthermore, collateral evolutions affect code related to library interfaces, and we have found in practice that the structure of such code is mostly determined by the constraints of the library rather than individual coding style. Thus, we expect that in the collateral evolution case, it will be possible to strike a balance between generality and simplicity, to create a transformation system that is useful in practice.

Another approach to aid in software evolution that has emerged in recent years is aspect-oriented programming (AOP) [6]. In this approach, new functionalities are expressed in a separate module, called an *aspect*, and woven into the original program at compile time or run time. Fiuczynski *et al.* are developing a variant of AOP, named C4, that allows expressing semantic patches with a different purpose than those described here, that of integrating new functionalities into Linux [4, 5]. Our work, focussing on collateral evolutions rather than new complete functionalities, is complementary to theirs, as once a C4 semantic patch has been created for a specific version of Linux, it is subject to collateral evolutions, just like any other Linux service-specific code. Furthermore, the technical goals of the two approaches are different, as we aim to transform the source code, to keep the code base up to date, as is currently done by hand in Linux, whereas they aim to create a new kind of pluggable module. Still, in previous work we have developed pluggable modules using some of the technology envisioned for SmPL [8].

Future steps

In this paper, we have presented the collateral evolution problem, and our proposed solution, the Coccinelle program transformation framework. More information can be found in our study of collateral evolutions in Linux device drivers [10] and our preliminary design of SmPL [9]. Currently, we are refining the design of the language and developing the corresponding transformation tool. Future work includes developing better tools for identifying collateral evolutions and tools to help in the creation of semantic patches. More information about the project is available at `http://www.emn.fr/x-info/coccinelle/`.

References

[1] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[2] C. Consel, J. L. Lawall, and A.-F. Le Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming*, 52:341–370, 2004.

[3] M. Erwig and D. Ren. Type-safe update programming. In *European Symposium on Programming (ESOP)*, 2003.

[4] 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.

[5] M. E. Fiuczynski. Better tools for kernel evolution, please! *;LOGIN:*, 30(5):8–10, Oct. 2006.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001.

[7] LWN. API changes in the 2.6 kernel series, Oct. 2005. http://lwn.net/Articles/2.6-kernel-api/.

[8] G. Muller, J. Lawall, J.-M. Menaud, and M. Südholt. Constructing component-based extension interfaces in legacy systems code. In *ACM SIGOPS European Workshop 2004 (EW2004)*, pages 80–85, Leuven, Belgium, Sept. 2004.

[9] 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.

[10] 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.

[11] 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.