Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 166 (2007) 47–62
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

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

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

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

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

Abstract

Collateral evolutions are a pervasive problem in large-scale software development. Such evolutions occur when an evolution that affects the interface of a generic library entails modifications, *i.e.*, collateral evolutions, in all library clients. Performing these collateral evolutions requires identifying the affected files and modifying all of the code fragments in these files that in some way depend on the changed interface.
We have studied the collateral evolution problem in the context of Linux device drivers. Currently, collateral evolutions in Linux are mostly done manually using a text editor, possibly with the help of tools such as `grep`. The large number of Linux drivers, however, implies that this approach is time-consuming and unreliable, leading to subtle errors when modifications are not done consistently.
In this paper, we propose a transformation language, SmPL, to specify collateral evolutions. Because Linux programmers are accustomed to exchanging, reading, and manipulating program modifications in terms of patches, we build our language around the idea and syntax of a patch, extending patches to *semantic patches*.

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

1 Introduction

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

In previous work [16], we have identified the phenomenon of *collateral evolution*, in which an evolution that affects the interface of a generic library entails modifications, *i.e.*, collateral evolutions, in all library clients. As part of this previous work, we have furthermore studied this phenomenon in the context of Linux device drivers.

doi:10.1016/j.entcs.2006.07.022

Collateral evolutions are a significant problem in this context because device drivers make up over half of the Linux source code and are highly dependent on the kernel and various driver support libraries for functions and data structures. This previous study identified a taxonomy of the kinds of collateral evolutions that are required in device drivers. These include changes in calls to driver support library functions to add or drop new arguments, changes in callback functions defined by drivers to add or drop required parameters, changes in data structures to add or drop fields, and changes in function usage protocols.

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

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

SmPL is a first step in a larger project to develop a transformation tool, Coccinelle, providing automated assistance for performing collateral evolutions. This assistance will comprise the SmPL language for specifying collateral evolutions and a transformation engine for applying them to device driver code. We expect that when the developer of a driver support library modifies the library's interface, he will create the corresponding semantic patch, relying on his understanding of the protocol for using the affected interface elements and the structure of typical driver code. He will then distribute the semantic patch to driver maintainers who will use it to update their drivers. Our goal is that the transformation process should be robust, and interactive when necessary. In particular, it should remain able to assist the driver maintainer even when an exact match of the rule against the source code is not possible, in the case of unexpected variations in driver coding style.

The rest of this paper is organized as follows. Section 2 describes a set of collateral evolutions that will be used as our running example. Section 3 illustrates how one of these collateral evolutions is expressed using the standard patch notation.