**Documenting and Automating Collateral Evolutions in Linux Device Drivers**

Yoann Padioleau Julia Lawall René Rydhof Hansen Gilles Muller

## 记录和自动附属演变Linux的设备驱动程序

为了解决新的需求和改善性能，Linux内部的库演变的非常迅速。这些演变，引发了一系列的Linux设备驱动程序的附属演变：对于每一个影响到API的改变，依赖这些API的驱动程序就必须随着更新，这种更新就是附属演变。手工的操作十分费时费力也不可靠，而且当这些更新不能被持续操作而发生中断时还易于出错。

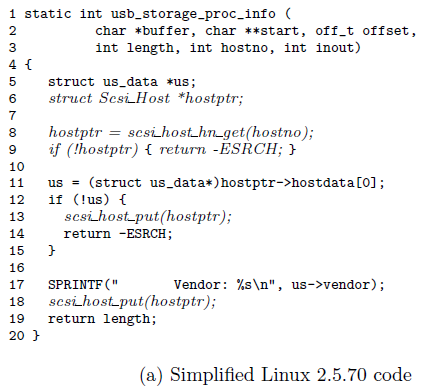
本论文作者提出了一个自动程序转换工具，Coccinelle, 用于记录和自动驱动的附属演变。这个工具使用一种基于补丁语法的语言来表示转换，扩展传统补丁为语义补丁（semantic patches）。Coccinelle 保留了驱动源程序的代码风格。这个工具对于库开发者【基于Emacs’ ediff的交互式模式帮助开发者书写补丁和验证补丁】（library developers）驱动维护者（driver maintainers）和一些用户（motivated user）【使用补丁】，十分有用。

Coccinelle 使用 SmPL（Semantic Patch Language）书写语义补丁，例如proc\_info 附属演变（collateral evolution）的一个补丁如下图2所示。图2描述了SmPL语义补丁描述的附属演变。这种语义补丁有传统语义补丁的格式，包括规则（rules）序列，以一对@@号分隔一些上下文信息为开始，然后规定了在这种上下文中应用的转换。在语义补丁的例子中，上下文信息声明了一组元变量。一个元变量可以匹配在其声明（标识符，整数表达式等）中的任何形式。转化被在一个传统的补丁文件中指定，如具有代码形式的术语需要转换。该术语用修饰符-和+的代码分别为删除和添加的形式表示。图2 的语义补丁文件包括两个规则，规则1是在给定的SCSI驱动文件中查找 proc\_info 函数的名字，规则2 是指定了应该被应用在这个函数的转换定义。此外，Coccinelle的转换引擎被同构指定组等价集合所参数化，这个同构在实施转换的时候会被考虑到，如指针判空操作，!x, x == NULL, 以及 NULL == x 虽然形式不同但在语义上是一样的，所以作者把这种同构定义为一种扩展补丁，如下所示。

@@ expression \*X; @@

X == NULL <=> !X <=> NULL == X

另外还有其他形式的同构，作者总共定义了39中同构，还有例如括号在代码同行或者换行，多空格少空格等等。



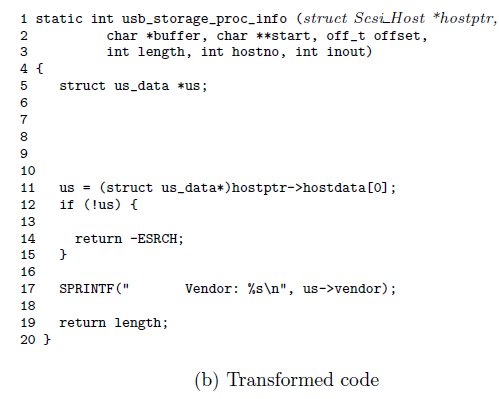
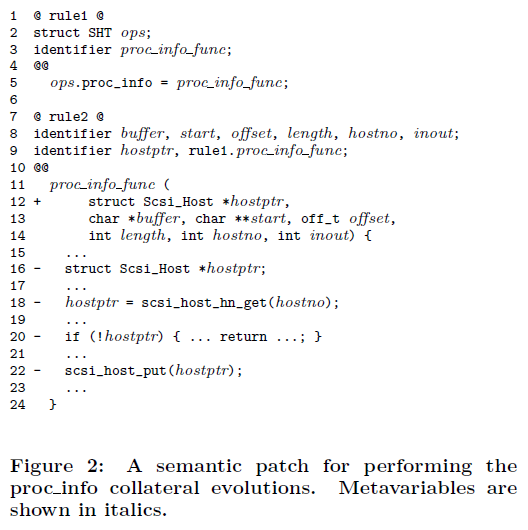
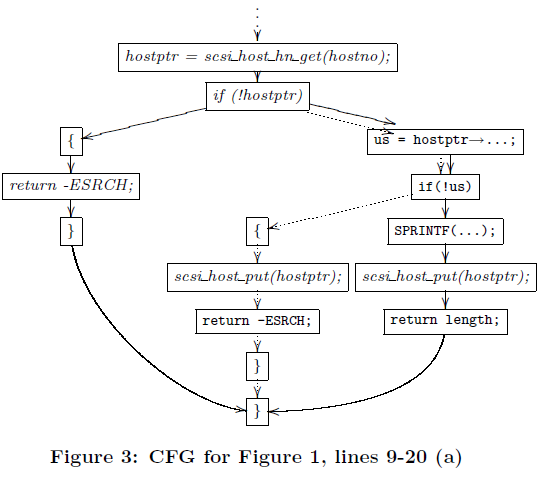


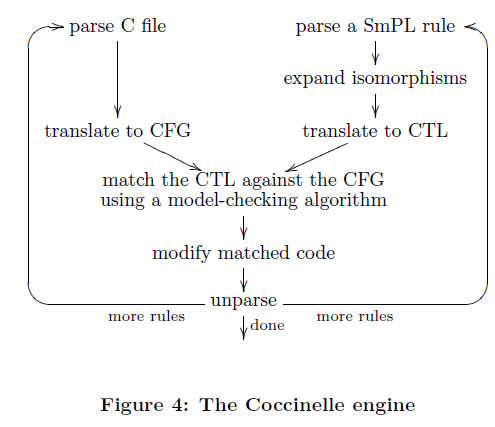
Figure: An example of collateral evolution, based on code in drivers/usb/storage/scsiglue.c



语义补丁描述了库所需的运行时操作的模式，该模式在所有驱动程序中都应该是相同的。由于这种相似，作者的Coccinelle工具使用驱动程序的控制流图（CFG）中的路径来描述。因此SmPL语义补丁中的序列形式匹配C语言中这种路径的序列形式。特别是，SmPL结构“...”是表示任意的CFG路径片段。图3是图1（a）的控制流图。图4是Coccinelle工作原理结构图，示出了由Coccinelle引擎所执行的主要步骤。在其设计的关键点是（i）与CPP应对，才能够保留原来的编码风格，（二），抽象句法和控制流变化，及（iii）有效地应用代码的转换策略。

SmPL 抽象句法和控制流变化是通过使用策略匹配语义补丁序列和代码控制流图的路径，为了实现这种匹配策略，作者使用了模型检测技术，这种技术就是基于图的[参考文献10]。在这种方法中，每一个顶级块的C源代码（例如，一个函数或宏定义）被转换成一个控制流图，这个图被作为模型，而SmPL语义补丁编译成一个时态逻辑公式（CTL [参考文献2 ]，有一些额外的功能，CTL是编译成汇编语言形式的）。公式对模型的匹配，是使用一个标准的模型检查算法的一个变种来实现[参考文献10 ]。匹配CTL公式对控制流图的结果是一个节点的集合，在这些点转换是必需的，语义补丁的片段匹配这些点以及元变量绑定。然后引擎就会根据修饰符-或+进行具体的转换。





为了高效的进行转换，Coccinelle使用了一些技巧进行优化。SmPL编译器大致收集关键字，如语义补丁中必须出现的被应用来匹配的函数和字段名字，例如图2中字段名字proc\_info 和函数名字scsi\_host\_hn\_get and scsi\_host\_put。然后引擎使用grep或者全文索引和搜索工具glimpse [参考文献15]来搜索包含至少一个关键字的文件。更细粒度的，SmPL编译器也收集原子模式，例如转换规则使用的场景中必须出现的函数调用，这些原子模式的检查在CFG创建过程中就被进行了，如果一些必须模式没有出现，则模型检查就不会被调用，这就起到了一定的优化作用。

作者将他们自己开发的工具patchparse [参考文献20]应用到Linux补丁中来检测常见的改变。从这些信息和受影响的驱动文件的学习，作者手动识别相关改变，并实现了语义补丁。

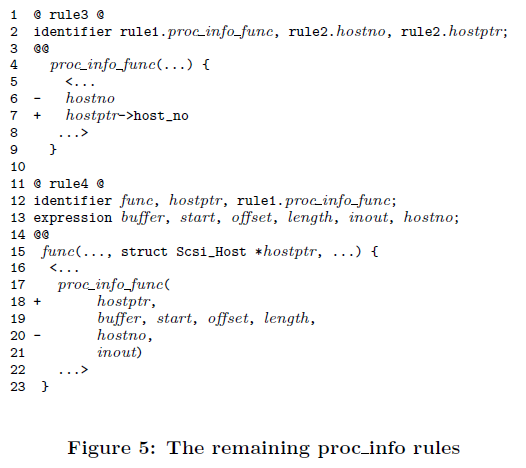


图5是解决同构现象的扩展语义补丁定义。操作符<... 和 ...> 包围的语句是不论在何时何地出现只要匹配就转换。

**局限性**：之一只能检测进程内部的控制流关系，不能检测数据流的和进程间的关系，例如一个函数可以变成一个宏定义，而这个宏可以在任何位置被定义；之二是一些结构的初始化，这个初始化位置可以在任意的合理位置；之三就是还不支持字符串内容的改变。