Enforcing the Use of API Functions in Linux Code

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

Gilles Muller
EMN/INRIA-Regal
France
Gilles.Muller@emn.fr

Nicolas Palix
DIKU, University of Copenhagen
Denmark
npalix@diku.dk

ABSTRACT

In the Linux kernel source tree, header files typically define many small functions that have a simple behavior but are critical to ensure readability, correctness, and maintainability. We have observed, however, that some Linux code does not use these functions systematically. In this paper, we propose an approach combining rule-based program matching and transformation with generative programming to generate rules for finding and fixing code fragments that should use the functions defined in header files. We illustrate our approach using an in-depth study based on four typical functions defined in the header file `include/linux/usb.h`.

Categories and Subject Descriptors

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

General Terms

Languages, Reliability

Keywords

APIs, Linux, bug-finding, bug-fixing

1. INTRODUCTION

A modular software design based on component encapsulation, where components interact via abstract interfaces, is well known to have many benefits. It improves readability, by allowing code to be expressed in terms of high-level concepts instead of implementation details. It improves correctness, by restricting operations on data to those that are well-defined. And it improves maintainability, by allowing implementations to be changed without requiring a global rewrite of the software. Modern languages such as Java provide facilities for enforcing encapsulation, by allowing methods and variables to be declared to have various degrees of restricted visibility. The C language, however, which remains widely used in operating systems and other low-level code, only provides very coarse-grained encapsulation via the `static` construct. Typically, all fields of all data structures are visible to all clients, making it possible to break encapsulation by performing data accesses directly.

Despite being implemented in C, the Linux kernel is structured in a very modular way, providing a collection of libraries for functionalities such as memory management, file systems, and various driver types. These libraries export interfaces via header files that typically define collections of high-level functions, either directly or as prototypes, for interacting with the library and manipulating relevant data. Nevertheless, complete encapsulation is not provided; the C language does not provide a means to enforce the use of these functions. This lack of encapsulation is particularly problematic because Linux is developed according to an open source model, with a large developer base. Not all of the developers who contribute to the Linux kernel have the same level of expertise in the many Linux APIs.

It seems unrealistic that the Linux kernel, or many other kinds of software with similar constraints, will be translated from C into a language that provides more encapsulation guarantees. Thus, it is necessary to check the code, and convert it to use API functions when needed. We focus on the many small functions defined in Linux header files that access fields in data structures, access bits in packed binary data, and wrap a generic function to produce a subsystem specific one. Although often trivial, these functions play an important role in ensuring the readability, correctness, and maintainability of the system. While a number of approaches has been developed for inferring API usage protocols [5, 7, 10, 14] and checking that they are followed [1, 2, 4, 7], none of the approaches of which we are aware directly address the problem of enforcing API usage in this context.

In this paper, we propose an approach to enforcing the use of API functions based on finding code fragments that match the body of an API function and replacing these code fragments by the corresponding API function call. For this, we build on the Coccinelle program matching and transformation engine [3, 12] that we have developed in previous work. Rather than requiring the user to manually write a rule for each API function, we take an approach based on generative programming. Specifically, we are extending the Coccinelle system to be able to generate new program matching and transformation rules based on patterns matched in existing code. We use this facility to scan Linux header files for certain types of API functions, and then generate rules for

ACP4IS'09, March 2, 2009, Charlottesville, Virginia, USA.

searching the Linux source tree for code fragments to replace by calls to these functions. This generative approach makes it possible to rapidly find API usage problems with no prior expertise in the APIs. Even though the patterns considered are not exhaustive, they are sufficient to find many non-uses of API functions in practice and can focus attention on APIs that do not seem to be well understood.

At present, this work is very preliminary, and thus we consider only the requirements posed by a single Linux header file, and a few simple types of API functions. Nevertheless, our previous work on Linux code, in the context of the development and evaluation of Coccinelle [7, 12], suggests that both the problem and the solution are much more general. The specific contributions of this paper are as follows:

- A study of the kinds of functions that are defined in the Linux file `include/linux/usb.h`. This header file describes the interface of the generic library for interacting with USB devices.
- An overview of our preliminary design of a facility allowing a Coccinelle program matching rule to generate a set of transformation rules based on the fragments of code that it matches.
- An assessment of the results obtained by applying our preliminary implementation to `include/linux/usb.h`.

The rest of this paper is organized as follows. Section 2 analyzes the contents of `include/linux/usb.h`. Section 3 presents our preliminary design of our extension to Coccinelle and our specifications for generating rules for detecting API usage problems. Section 4 evaluates our results. Section 5 presents related work. Section 6 concludes.

2. CASE STUDY: USB.H

The file `include/linux/usb.h` describes the interface of the generic library for interacting with USB devices. It is one of the most widely-used device-specific header files in Linux, being included in over 300 Linux kernel source files.[1] Many of these files are in the directory `drivers/usb`, and are thus watched over by USB experts. As shown in Figure 1, however, over half of these files are in other directories, where the developers and maintainers may have less expertise in USB details. The file `include/linux/usb.h` came to our attention because of seeing a patch that introduced a call to an API function defined in this file.[2] We subsequently manually wrote a Coccinelle rule to find and fix the non-use of a subset of the API functions defined in `usb.h` in over 40 Linux files.

The file `include/linux/usb.h` is 1751 lines of code, including comments and blank lines. It contains 18 structure type definitions, 75 macro definitions, 57 prototypes for external functions, and 38 function definitions. Our approach focuses on the function definitions. 8 of the function definitions either return 0 or do nothing. These functions are provided for features that are not supported, and are of no interest in our case. Another 8 of the function definitions perform some complex operations, often amounting to initializing a set of structure fields. Finally, 22 of the function definitions either initialize a single structure field, return the value of an expression, or call some other function. Representative examples are shown in Figure 2. These functions are the focus of this work.

[1]Linux-next version 20081219, git code 8dba8f585debbba97-86aa6fccbe863998d865891. Subsequent references to Linux refer to this version, unless otherwise noted.

[2]Git code 2fe3f7501af820f98c0f98aac140aa24776d175d.

drivers/usb	136	drivers/isdn	8	drivers/mtd	1
drivers/media	55	drivers/uwb	5	drivers/w1	1
drivers/net	36	drivers/bluetooth	4	drivers/watchdog	1
drivers/staging	27	drivers/input	3	fs	1
sound/usb	12	drivers/block	1	include/linux	1
drivers/hid	11	drivers/i2c	1	security	1

Figure 1: Distribution of the use of `usb.h` across the directories of the Linux kernel source tree

```
static inline void
usb_set_intfdata(struct usb_interface *intf, void *data)
{
 dev_set_drvdata(&intf->dev, data);
}

static inline void
usb_mark_last_busy(struct usb_device *udev)
{
 udev->last_busy = jiffies;
}

static inline int
usb_endpoint_xfer_isoc(const struct usb_endpoint_descriptor *epd)
{
 return ((epd->bmAttributes &
          USB_ENDPOINT_XFERTYPE_MASK) ==
         USB_ENDPOINT_XFER_ISOC);
}

static inline int
usb_endpoint_is_isoc_in(const struct usb_endpoint_descriptor *epd)
{
 return (usb_endpoint_xfer_isoc(epd)&&usb_endpoint_dir_in(epd));
}
```

Figure 2: Extracts of `usb.h`

3. OUR APPROACH

Our goal is to find fragments of code in the Linux kernel source tree that match the bodies of the functions defined in `usb.h` and to replace these fragments by calls to these functions. To do this, we propose a three step approach. The first step collects information about the definitions of various types of API functions. The second step uses this information to generate rewrite rules that search for code matching the bodies of the identified API functions and replace this code by an API call. The third step applies the generated rules to the code base, to check for and enforce the use of the API functions. Each of these steps is carried using Coccinelle.

We first give a brief overview of Coccinelle, using as an example a rule that carries out the first step. We next consider the second step, how to use Coccinelle to generate rules for introducing the use of API functions. Section 4 presents the results of the third step, applying the generated rules to the Linux kernel source tree.

3.1 Overview of Coccinelle

Coccinelle is a tool for performing program matching and transformation on (unpreprocessed) C source code according to user-provided specifications. Specifications are defined using a patch-like notation, and are thus very close in struc-

```
@r1@ identifier f; parameter list ARGS; expression E1, E2;          1
identifier fld; @@                                                  2
                                                                    3
f(ARGS) { E1->fld = E2; }                                           4
```

Figure 3: A semantic patch rule identifying a particular type of API function

```
@@ struct usb_device *udev; @@                                      1
                                                                    2
- udev->last_busy = jiffies;                                        3
+ usb_mark_last_busy(udev);                                         4
```

Figure 4: A specification of the transformation introducing calls to `usb_mark_last_busy`

ture to the code to be matched and transformed. Coccinelle specifications are, however, more generic than the traditional line-based syntax-driven patches, and thus we refer to them as *semantic patches*. In this paper, the primary aspect of genericity that we are concerned with is the ability to abstract over the subterms of a code fragment using *metavariables*, which represent arbitrary terms of a particular syntactic category, *e.g.*, an expression, statement, or parameter list, or expression type, *e.g.*, an integer, pointer, or structure.[3] Metavariables allow one code-like pattern to match at many places in a software code base.

A semantic patch consists of a collection of rules, each of which performs a single matching and transformation task. A rule may inherit metavariables bound by previous rules, allowing information to flow between them. An example of a semantic patch rule `r1` is shown in Figure 3. A rule consists of two regions. The first, delimited by a pair of `@@`s, of which the first can optionally contain the name of the rule, declares the metavariables (lines 1-2). The rest is an arbitrary term, which may contain one or more occurrences of the metavariables, describing the code to be matched. Although not shown in Figure 3, lines of code can be annotated with `-` to indicate that they should be dropped and with `+` to indicate that they should be added, in both cases exactly as done in a patch file.

The goal of the rule `r1` in Figure 3 is to find functions that only serve to initialize a single structure field, such as the function `usb_mark_last_busy`, shown in Figure 2 (lines 7-11). The rule declares an `identifier` metavariable *f* to represent the function's name, a `parameter list` metavariable *ARGS* to represent the function's parameter list, `expression` metavariables *E1* and *E2* to represent the structure and the initial value, respectively, and an `identifier` metavariable *fld* to represent the name of the field to be initialized. The rest of the rule has the form of a definition of a function having the desired structure, using the metavariables appropriately. When applied to the Linux kernel, this rule simply matches the pattern against the code and binds the metavariables accordingly; no transformation is specified. This rule is thus suitable for using in a larger semantic patch that inherits the set of metavariable bindings derived from each matching instance to perform some other action.

3.2 Rule generation

The rule `r1` matches the function `usb_mark_last_busy`, defined in Figure 2, with *f* bound to `usb_mark_last_busy`, *ARGS* bound to `struct usb_device *udev`, *E1* bound to `udev`, *fld* bound to `last_busy`, and *E2* bound to `jiffies`. From this information, we would like to generate a rule that matches `udev->last_busy = jiffies;` and replaces it by a call to `usb_mark_last_busy`, essentially the rule shown in Figure 4. Thus, in general, the generated rule should match the pattern *E1->fld = E2;*, where *E1*, *fld*, and *E2* are instantiated according to the bindings obtained by rule `r1`.

The generative rule begins as follows, to indicate that it is a rule that generates other rules:

```
@generated@                                                         1
```

As before, we next specify the metavariables. It is possible both to inherit metavariables from previous rules, which will be used in generating the current rule, and to declare local metavariables, which will be copied into the generated rule. In this simple case, we do not need any local metavariables, and so we just add declarations to inherit metavariables from `r1`. Inherited metavariables are referenced in the declaration as $r.n$, where r is the rule name and n is the name of the inherited variable. In the body of the rule, the name of the inherited variable n is used directly. In our case, we obtain:

```
@generated@ expression r1.E1, r1.E2; identifier r1.fld, r1.f; @@    1
```

Finally, we specify the pattern to generate, which is essentially as follows, with the metavariables instantiated by their values:

```
- E1->fld = E2;                                                     1
+ f(???);                                                          2
```

To complete the rule, we must determine what the arguments "???" to the API function *f*, *e.g.*, `usb_mark_last_busy`, should be. First, however, we consider a related issue: how to treat the names `udev`, `last_busy`, and `jiffies` found in the definition of `usb_mark_last_busy`. `last_busy` is the name of a structure field and `jiffies` is a global variable, so they should appear as-is in any matched code.[4] `udev`, however, is a variable local to `usb_mark_last_busy`, and indeed it might not be the case that every initialization of the field `last_busy` of a `usb_device` structure gives the structure the name `udev`. Our solution is to consider the parameters of the matched function to be metavariables, of either the parameter type, or if the parameter type is `void *`, then of type `expression *`. We furthermore use these metavariables as the arguments to the added function call. For this purpose, in the context of rule generation, *ARGS* is a special variable, that should be bound in step 1 as the list of parameters of the matched function, but is inherited in the generated rule as a list of argument expressions, amounting to the parameter declarations with the type information removed. This approach assumes that all of the parameters of the matched function appear in the function body, and thus in the pattern of the

[3] Unlike in AspectJ, it not possible to constrain the textual representation of the code matching a metavariable, *e.g.*, by specifying part of an identifier name. Our study of Linux code suggests that such constraints are not reliable in this setting.

[4] It is possible that a global variable is shadowed by a local variable declaration at the point of a match. Currently, we simply assume that this is not possible, and indeed it seems rare in Linux that both a global variable and a local variable have the same name. Nevertheless, in future work, we should extend the approach to protect against such confusion.

```
@r1@ identifier f,fld; parameter list ARGS; expression E1,E2; @@
f(ARGS) { E1->fld = E2; }

@generated@
expression r1.E1, r1.E2; identifier r1.fld, r1.f;
expression list r1.ARGS; @@

- E1->fld = E2;
+ f(ARGS);
```

Figure 5: A semantic patch for generating a rule from a particular type of API function

generated rule. When this is not the case, something else has to be done to create the arguments of the generated function call. We leave this issue to future work.

The complete semantic patch for generating a rule matching a field initialization is shown in Figure 5. When this semantic patch is applied to the definition of the function `usb_mark_last_busy` it essentially generates the rule shown in Figure 4. This rule, however, is not constrained enough to always apply safely; we consider this issue below.

Besides `usb_mark_last_busy`, Figure 2 illustrates functions that simply call another function (`usb_set_intfdata`) and functions that simply return the value of an expression (`usb_endpoint_xfer_isoc` and `usb_endpoint_is_isoc_in`). The rule-generating semantic patches for these cases are similar to the one for a function that just initializes a structure field, but slightly more complex, because there are some cases in which it is either not desirable to create the rule, *e.g.* there is no point to create a rule for a function that always returns a constant such as 0, because there is no way to know which 0's should be replaced by calls to this function and which should not, and there are some cases in which the generated rule should not apply, *e.g.*, a rule generated from a function that just accesses a location can be used when the location is used as an R-value, but not when the location is used as an L-value. The complete semantic patch for generating rules for all three kinds of functions is available at our website.[5]

3.3 Refinement of the generated rules

The simple replacement expressed in the generated rule shown in Figure 4 is not adequate to ensure that the transformation is performed correctly. First, it may perform the transformation in a file where `usb.h` is not included and thus `usb_mark_last_busy` is not defined. Second, because Coccinelle by default processes the header files that a C file includes, when a C file does include `usb.h`, this simple replacement will match the body of the definition of the API function itself, creating a function that loops endlessly. To avoid both of these issues, Coccinelle embeds the generated rule into a larger semantic patch that performs appropriate checks before performing the transformation.

The complete semantic patch generated from the definition of `usb_mark_last_busy` is shown in Figure 6. The first rule detects that the header file on which the rule generation is based is included in the C file being processed. The second rule records the position of the definition of `usb_mark_last_busy` if it is present in the file. The final rule embeds the transformation derived from the definition of `usb_mark_last_busy`. As indicated by its first line (line 7),

[5]http://www.emn.fr/x-info/coccinelle/acp4is.cocci

```
1  @header@ @@
2  #include <linux/usb.h>
3
4  @same depends on header@ position p; @@
5  usb_mark_last_busy@p(...) { ... }
6
7  @depends on header@
8  position _p!=same.p; identifier _f; struct usb_device *udev; @@
9
10   _f@_p(...) { <+...
11     // match the pattern below 1+ times in the function body
12 -   udev->last_busy = jiffies;
13 +   usb_mark_last_busy(udev);
14   ...+> }
15
```

Figure 6: The semantic patch generated for detecting the non-use of `usb_mark_last_busy`

	current calls	needed calls	calls updated	% calls updated
usb_set_intfdata	303	4	4	100%
usb_mark_last_busy	15	6	6	100%
usb_endpoint_xfer_isoc	8	11	3	27%
usb_endpoint_is_isoc_in	14	1	0	0%

Figure 7: Results of applying the generated semantic patches to the Linux kernel source tree

this rule only applies if the rule header previously applied successfully in this file, ensuring that the required header file has been included. As indicated by the constraint on the position variable *_p* (line 8) and the match of the transformed function *_f* (line 10), a function is only transformed if its position is different than that of a definition of `usb_mark_last_busy`. With these constraints, the generated rule can be applied safely throughout the Linux kernel.

4. EVALUATION

Applying the semantic patches for generating rules from header file functions that either initialize a single structure field, return the value of an expression, or call some other function produces over 5400 rules when considering all of the header files in the Linux kernel source tree. On a HP ProLiant server with two 3GHz quad-core Xeon processors and 16GB memory, where we use only one core, generating these rules requires around 24 minutes. Due to time and space constraints, we focus on the rules generated from the functions shown in Figure 2. Figure 7 shows the number of calls to each of these functions currently in the Linux kernel, an estimate of the number of other code fragments that should use these functions based on manually searching for relevant code fragments, and the number and percentage of these code fragments that are updated by the generated rules. Applying these rules to the entire Linux kernel, again using only one core, takes around 8 seconds. The generated rules produce no false positives; all of code fragments that are updated are updated correctly.

For `usb_set_intfdata` and `usb_mark_last_busy`, the generated rules update all code fragments in the Linux kernel source tree that should use the API functions. When looking at the results for the rule for `usb_set_intfdata` we furthermore noticed a bug in the existing Linux code, as a function in the file drivers/media/video/zr364xx.c calls both `usb_set_intfdata` and `dev_set_drvdata` on the same data, thus

unnecessarily duplicating work. The rules generated from the remaining functions, however are less successful; there are a number of false negatives. The reasons for these failures highlight some inadequacies of the current approach that must be addressed in the future development of this work.

As shown in Figure 2, the function `usb_endpoint_xfer_isoc` is defined in terms of the macros `USB_ENDPOINT_XFER_TYPE_MASK` and `USB_ENDPOINT_XFER_ISOC`. In the Linux source code, however, sometimes these macros are not used, being replaced by the constants to which they expand. Thus, the generated rule does not match. One approach would be to create a second set of rules based on the result of applying the C preprocessor to the header file. This, however, would not be sufficient in the general case where macros and their expansions can be mixed. A better solution would be to generate rules that first replace these constants by the associated macros. Generating rules from macro definitions is, however, challenging, because of the lack of available type information, making it unclear which of a set of macros that expand to given value should be chosen.

A similar issue causes the total failure of the rule for `usb_endpoint_is_isoc_in`. Here, the dependence is on API functions rather than macros. As shown in Figure 2, the definition of `usb_endpoint_is_isoc_in` refers to the functions `usb_endpoint_xfer_isoc` and `usb_endpoint_dir_out`. Currently, there is no occurrence of these two functions that matches the required pattern in the Linux kernel source tree; the generated rules for these functions have to be applied first to create them. To address this issue, we envisage an auxiliary tool that identifies dependencies between the generated rules and orders them accordingly. Such a tool could also apply to rules generated from macro definitions, if available, thus addressing the issues arising in both cases.

Finally, a further issue occurs in the case of `usb_endpoint_xfer_isoc` due to the possibility of negation. As shown in Figure 2, `usb_endpoint_xfer_isoc` returns the value of an equality test, implying that the same function can be used in place of an inequality by simply negating the result of the call. Our generated rule is not able to express this possibility, because the rule-generating semantic patch treats the argument of the return as a single expression, without analysing its internal structure. A solution would be to create a rule-generating semantic patch that is specific to functions that return the result of an equality test, which would then generate two rules, one introducing a call to the matched function on finding an equality test and another introducing a negated call to the matched function on finding an inequality test. Such a rule-generating semantic patch would likely match other API functions as well.

We have manually written a semantic patch based on the definitions of `usb_endpoint_xfer_isoc` and `usb_endpoint_is_isoc_in`, where the rule for `usb_endpoint_xfer_isoc` matches either the given macro or its expansion, and the semantic patch is ordered such that the rule for `usb_endpoint_xfer_isoc` precedes the rule for `usb_endpoint_is_isoc_in`. This semantic patch was sufficient to update all but two of the remaining cases. The last two cases involve a macro that prevents Coccinelle from obtaining adequate type information about the structure expression. In studying the results of the semantic patch, we furthermore identified some related code in which the wrong macro appears to be used. We have submitted our results as patches to the appropriate Linux maintainers, and some of these patches have been accepted (none of them have been rejected). As a side effect of our work, the Linux maintainers have reconsidered the placement of `usb_endpoint_xfer_isoc`, `usb_endpoint_is_isoc_in`, and other related functions in `usb.h`, and have moved these functions into the file `include/linux/usb/ch9.h`, which is the header file that defines the various macros and structure types that are used by these functions. Sometimes finding erroneous code can have unexpected consequences.

5. RELATED WORK

As mentioned previously, a number of approaches have considered the problem of automatically detecting API usage protocols, typically involving sequences of two or more function calls [5, 7, 10, 14]. Here we consider functions that are typically not controlled by such protocols, but instead may be used independently any number of times. Our own previous work on detecting API protocols using Coccinelle [7] involved the use of an initial semantic patch to collect some information about an API protocol followed by the use of semantic patch templates that could be instantiated with respect to function names or other information collected in the first phase. The instantiation, however, was done in an ad hoc manner, using tools external to Coccinelle. Furthermore, that work did not consider the specific case of API functions contained in header files.

By allowing a semantic patch to generate other semantic patch rules, we have take a step towards developing a multi-level semantic patch language. Multi-level languages have been extensively investigated at a theoretical level [11], and in the context of languages such as C [13] and ML [15]. A significant issue in the context of multi-level languages is the availability of values at various levels. In particular, variables bound in earlier levels can only be used in later levels if their values can be represented as code, and variables bound in later levels should not be used in earlier levels. In our case, metavariables bound to positions cannot be represented as code, and Coccinelle prohibits such variables from being used in generated rules. The language also prohibits generated rules from having names, preventing rules at any level from inheriting their variables. This, however, is an overly conservative solution, and we will consider how to allow inheritance within rules at the same level in future work. Finally, our current approach provides only two levels, but it would be natural to consider extending the approach to more.

Refactoring is a type of program transformation that has the goal of restructuring the code but preserving its semantics [6]. In the context of refactoring, tools have been developed to detect code clones, to abstract common code fragments into a function, and to merge identical function definitions into a single function [8, 9]. Of these, clone detection is the most time-consuming and the most liable to suffer from false negatives and false positives. Our approach can be viewed as a means of focusing the clone detection on particular code fragments that have been designated as having a conceptual interest by the software designer. Furthermore, in our approach the structure of the API function definition and the structure of the code that is to be replaced by a call to the API function are specified separately and under control of the programmer, thus giving more flexibility than with a fixed strategy that is built into the refactoring engine.

6. CONCLUSION

Linux header files contain many API functions that are simple in their behavior, but nevertheless are critical to the readability, correctness, and maintainability of the Linux kernel code. Currently, these API functions are not used as systematically as they should be. In this paper, we have presented an approach based on generative programming for finding and correcting code in which these API functions are not used. In our evaluation based on the file `include/linux/usb.h`, we have found that in some cases the approach outlined here is sufficient to update all of the relevant code correctly. We have furthermore outlined how to address the remaining cases in future work.

7. REFERENCES

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07*, pages 43–48, San Diego, CA, June 2007.

[2] 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.

[3] J. Brunel, D. Doligez, R. R. Hansen, J. Lawall, and G. Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, Jan. 2009.

[4] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, Oct. 2000.

[5] D. R. Engler, D. Y. Chen, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 57–72, Banff, Canada, Oct. 2001.

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

[7] J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, and G. Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. Technical Report 08/1/INFO, Ecole des Mines de Nantes, Nantes, France, 2008.

[8] H. Li and S. Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming (TFP)*, Nijmegen, The Netherlands, May 2008.

[9] 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 Francisco, CA, Dec. 2004.

[10] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315, Lisbon, Portugal, 2005.

[11] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[12] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, Mar. 2008.

[13] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.

[14] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *29th International Conference on Software Engineering*, pages 240–250, Minneapolis, MN, USA, 2007.

[15] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.