On the Origins of Coccinelle

Julia Lawall
Inria, Paris, France

Abstract

Coccinelle is a program-transformation system for C code. It has been under development since 2005 and has been extensively used on the Linux kernel. The design of Coccinelle was inspired in part by the author's previous experience in using Stratego/XT, developed by Eelco Visser. This paper reflects on some of Coccinelle's design choices and their relation to Eelco Visser's work.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases Linux kernel, Coccinelle, Stratego/XT, program transformation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.18

Supplementary Material *Software (Source Code)*: `https://github.com/coccinelle/coccinelle`
archived at `swh:1:dir:94194d5e17db6802aeb51a826179b95a6f4f1808`

Acknowledgements The author wishes to thank Gilles Muller for his collaboration on the design, application, and maintenance of Coccinelle. The author would also like to thank the anonymous reviewers for their helpful feedback, particularly on historical matters, and Olivier Danvy for help in improving the clarity of the presentation.

1 Introduction

Program transformation has a long history [4, 20, 33]. Anyone who loves to write code also, somewhat ironically, begins to wonder whether a tool could write code for them, or at least perform the repetitive code transformations that are inevitably required as design choices change and a system evolves. This wondering has led generations of researchers to investigate the design of tools to automate various kinds of program transformations. Still, many of these approaches have been designed around toy languages or required users to express transformations at the abstract-syntax tree level, and are thus not practical for use on real, large software projects.

In 2004, the author, with Gilles Muller, began to investigate the problem of how to migrate Linux kernel device drivers from Linux kernel version 2.4 to Linux kernel version 2.6. At that time, the even numbered versions of the Linux kernel were considered to be *stable* releases, and were maintained with bug fixes, in parallel to the odd numbered versions, namely Linux kernel version 2.5, in which new features were integrated. Linux 2.4 was first released in January 2001 and Linux 2.6 in December 2003, amounting to a substantial time lapse, with many intervening changes across the code base. Manually updating code, such as device drivers maintained outside the Linux kernel code base, for use with Linux version 2.6 could be a substantial challenge. Our study of changes performed on device drivers in the Linux kernel source tree [23] showed that the changes required could range from simple refactorings [9], such as renaming a function or reorganizing a data structure, to scattered changes across a function or file, potentially depending on some properties of the code context. Our goal was thus to devise a transformation language that would be easy for Linux kernel developers to use in automating all such changes.

When we started this investigation, the idea of using tools to generate, analyze, and transform systems code was already attracting interest. The Devil domain-specific language for generating kernel-device interface code from high-level specifications was proposed in 2000 [19], with a follow-up paper in 2001 investigating the robustness of the generated code [26]. Metal [7] was proposed in 2000 for systematically searching Linux and OpenBSD code for patterns of code corresponding to bugs, and was followed up with strategies for inferring such patterns from analysis of a code base [8, 11]. AspectC [5] appeared in 2001, targeting the instrumentation of FreeBSD with pluggable cross-cutting strategies for prefetching data from the disk. CIL [20], published in 2002, offered a parser and visitor for processing C code facilitating the development of analyses and transformations of C code at the abstract-syntax tree level. Some of these approaches, such as Metal and CIL, allow finding complex patterns of code and, in the case of CIL, transforming them. But they require programming-language-specific expertise, in automata and abstract-syntax trees, that are not within the comfort zone of the typical Linux kernel developer. None of these approaches enables systems code developers to easily create fine-grained transformation rules for the specific needs of porting Linux kernel code from an older version to a more recent one. Instead, it was Eelco Visser's Stratego/XT [1], a transformation system designed for general-purpose code written in languages such as Java, that provided a guiding light.

The rest of this paper illustrates some recurring code changes that were performed in Linux 2.5 and Linux 2.6, gives a brief overview of Stratego/XT and how it inspired our work, and concludes with a short presentation of our transformation tool for C code, Coccinelle [22]. Coccinelle was first released in open source in 2008. As of October 2022, Coccinelle has been used in over 9000 commits to the Linux kernel. More information about Coccinelle is available in a range of research papers on its design and application [3, 13, 14, 15, 16, 18, 24, 27], and at the Coccinelle web site.[1]

2 Some examples of Linux kernel changes

This section presents some widespread changes that were performed in Linux 2.5 or 2.6 that illustrate some important requirements for Coccinelle. The examples come from the `history` tree of the Linux kernel.[2] The change extracts, expressed as Unix *patches* [17], have been simplified for readability and to focus on the most relevant changes.

2.1 A new argument for `end_request`

Our first example illustrates a straightforward change that only requires considering a single function call. Nevertheless, it illustrates what can go wrong when developers make such changes manually or using tools such as search and replace that are not aware of the programming-language syntax.

Prior to Linux 2.5.22, the function `end_request` always worked on the request stored in the global variable `CURRENT`. In Linux 2.5.22, with the goal of eliminating `CURRENT`, the function was reorganized to take the request as an argument. The first step in the reorganization was thus to add `CURRENT` as a first argument to the existing calls, as illustrated by the patch in Figure 1.

```
@@ -929 +929 @@
- end_request(res);
+ end_request(CURRENT, res);
```

Figure 1 Change to `end_request` in `drivers/mtd/nftlcore.c` in commit 4fe6433a5d9e.

[1] https://coccinelle.gitlabpages.inria.fr/website
[2] git://git.kernel.org/pub/scm/linux/kernel/git/history/history.git

This straightforward change affected 40 files. Still, it was noted in the original work on Coccinelle [22] that the change was also incorrectly applied to a call to the unrelated function `swimiop_send_request`, showing the importance of a change-automation tool being aware of token boundaries in the programming language.

2.2 Changes in the usage of `usb_register_dev` and `usb_deregister_dev`

Our second example illustrates a case where constructing the changed code requires extracting information from other parts of the affected file.

In Linux 2.5.25, the function `usb_register_dev` lost its first argument and gained two new arguments, as illustrated by the patch in Figure 2. The new values are not part of the original call, nor in the adjacent code. Instead, they are the values stored in some members of the structure whose value was the original first argument.

```
@@ -815 +815,2 @@
- retval = usb_register_dev(&usblp_driver, 1, &usblp->minor);
+ retval = usb_register_dev(&usblp_fops, USBLP_MINOR_BASE, 1,
+                           &usblp->minor);
```

Figure 2 Change to `usb_register_dev` in `drivers/usb/class/printer.c`, in commit 26f8beab467e.

At the same time, the function `usb_deregister_dev` lost its first argument, as illustrated by the patch in Figure 3. This change is straightforward, requiring only information that is part of the original function call.

```
@@ -375 +375 @@
- usb_deregister_dev (&usblp_driver, 1, usblp->minor);
+ usb_deregister_dev (1, usblp->minor);
```

Figure 3 Change to `usb_deregister_dev` in `drivers/usb/class/printer.c`, in commit 26f8beab467e.

These changes affected 9 Linux kernel files, and show the importance of being able to collect information from across the definitions in a file, or even the entire code base.

2.3 Introduction of `kzalloc`

Our final example illustrates a case in which program control-flow has to be taken into account.

In Linux version 2.6.14, the function `kzalloc` was introduced, to replace the composition of a call to the Linux kernel memory allocator `kmalloc` and a zeroing call to `memset`, as illustrated by the patch in Figure 4.[3] Unlike our other examples, this change is not obligatory, because the functions `kmalloc` and `memset` remain unchanged in the Linux kernel. Still, making the change results in code that is more concise and readable, since the zeroing is immediately apparent at the allocation site.

[3] This change was overlooked over the years, and was made by the author in 2022: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3c0e3ca6028b`

```
- request = kmalloc(16, gfp_mask);
+ request = kzalloc(16, gfp_mask);
  if (!request)
    return -ENOMEM;
  urb = usb_alloc_urb(0, gfp_mask);
  if (!urb) {
    kfree(request);
    return -ENOMEM;
  }
- memset(request, 0, 16);
```

Figure 4 Introduction of `kzalloc` in `drivers/usb/net/zd1201.c`, in commit 3c0e3ca6028b.

This change raises a number of challenges. Commonly, `kmalloc` and `memset` are only separated by a conditional checking whether the allocation failed. Other intervening code, however, is possible, as shown in Figure 4, raising the need to adapt to different scenarios. Both `kmalloc` and `memset` are very common in the Linux kernel; the change should only be performed if the `memset` occurs in the control flow whenever the allocation succeeds and before the allocated memory is used. Furthermore, it is desirable for the amount of memory that is allocated and zeroed to be the same, as will be the case after the introduction of `kzalloc`.

3 Stratego/XT

Stratego/XT provides a language and a toolset for program transformation [1]. It grew out of Visser's work on rewriting in the late 1990s [31, 32]. Stratego/XT offers the ability to write complex transformations as a composition of a *strategy*, which is used to select terms to transform, and one or more transformation rules, which describe what changes to perform. Strategies describe how to navigate around an abstract-syntax tree to identify terms to consider for transformation. Transformation rules describe how to decompose an existing term, and use the components to build a new one. Stratego/XT is programming-language-independent. Internally, it adopts the *Aterms* of van den Brand et al. [28], that consists of a constructor name and a sequence of arguments. Stratego/XT uses ATerms to describe an abstract-syntax tree.

Writing transformation rules in terms of abstract-syntax trees requires the user to be aware of the constructor names and corresponding arguments that describe the various kinds of terms of interest. Obtaining this information may require studying documentation or even a language implementation, and the resulting transformation rules are verbose, since, e.g., even representing a simple variable `x` requires both the constructor for variable references and that constructor's argument list. The Stratego/XT user, however, does not interact with ATerms directly. Instead, Stratego/XT uses Syntax Definition Formalism (SDF), developed in Eelco Visser's PhD thesis [30], to provide parsers for various languages. A transformation is expressed as a fragment of concrete syntax, parameterized by *metavariables*, that is to be transformed into another fragment of concrete syntax, possibly referencing the metavariables mentioned in the former fragment. A metavariable, as previously proposed by van Deursen et al. [29], is a variable representing an arbitrary term of a particular syntactic category: expression, integer constant, etc. Metavariables are declared by prepending `~` to the variable name, but there are some predefined metavariables, such as `e` for an expression, or `i` for an integer constant. Using these predefined metavariables makes the fragments in the transformation rule look entirely like ordinary code, which makes the transformation to be performed instantly recognizable to the software developer.

At the time when we were first designing Coccinelle, Stratego/XT mainly supported Java code; a port for C code existed, but feedback from Eelco Visser suggested that it did not support enough of the C language to be reliably applied to the Linux kernel. In some experiments that we had performed in the context of another project, using Java, we also found that the strategies, while elegant, could also be complex to use, perhaps requiring more intuition about code as an abstract-syntax tree than would be typical of most Linux kernel developers. Nevertheless, the combination of fragments of concrete syntax, abstracted by metavariables, particularly metavariables that do not require any specific annotation within the fragments, appeared to be a powerful approach around which to design a user-friendly transformation language for real world software.

4 Coccinelle

Inspired by the ease of specifying transformation rules in Stratego/XT in terms of concrete syntax, in 2005, we began the design of Coccinelle. The initial idea was to propose a transformation language in which rules were explicitly composed of a phase for the collection of information from the source code, followed by a phase for the processing of that information. This design was motivated by Stratego/XT's strategies. Our postdoc, Yoann Padioleau, however, pointed out that this design still relied too much on a programming-languages point of view. To really provide something that would be usable by Linux kernel developers, we should directly follow their existing habits. Linux kernel developers are focused on lines of source code, and exchange and reason about changes in terms of *patches* [17], i.e., extracts of the source code in which lines to remove are indicated by a `-` at the beginning of the line, and lines to add are indicated by a `+`. Patches are easy to create from source-code changes, using the Unix `diff` tool [17], and easy to understand, because they contain the source code that the developer is already familiar with. They have the disadvantage, though, that each patch is completely tied to specific locations in the source code, due to the use of file names, line numbers, and the specific variable names, whitespace, etc. in the changed code and its context. To create Coccinelle's *semantic patch language*, SmPL, we considered how to make the patch notation more generic.

A semantic patch consists of a sequence of one or more *rules*, each of which describes code to remove and add. A SmPL rule has two parts: first a list of metavariables, surrounded by a pair of `@@`, that can be used in the second part, a pattern, describing the code to match and transform. Rather than using `~` and reserved names, as in Stratego/XT, SmPL metavariables are declared explicitly with their syntactic categories, such as `expression` or `statement`,[4] indicating the kind of code term that they can match. A pattern is a code fragment parameterized by metavariables. Lines in this fragment can be annotated with `-` and `+`, indicating code to remove and add, respectively. SmPL contains operators for describing (intraprocedural) program control-flow and expression types, i.e., some minimal semantic properties. We illustrate the main features of SmPL by revisiting the examples of Section 2.

[4] As fans of statically typed languages, we considered such declarations to be important, but we were told later by some systems code developers with no formal programming-languages training that they found the need to choose the syntactic category of a metavariable to be a burden, because they were not familiar with the terminology. We added a generic syntactic category `metavariable`, but this syntactic category cannot always be used, because the metavariable syntactic categories are sometimes essential to avoid parser conflicts.

4.1 A new argument for `end_request`

To make the change for `end_request` (Figure 1), it is necessary to match the existing call, remove it, and replace it with one that has `CURRENT` as an extra first argument. The form of the original argument is not important, and so we abstract over this argument with a metavariable. Note that the resulting pattern is identical to the code found in the example change in Figure 1. As the argument was already a simple variable in the code of Figure 1, it suffices to abstract over this variable.

```
@@ expression res; @@
- end_request(res);
+ end_request(CURRENT, res);
```

Some variations on the above semantic patch are possible. First, a function call, such as the call to `end_request`, need not appear at the top level in a statement. To match the call as an arbitrary subexpression, the trailing semicolon can be omitted. The resulting semantic patch will match a function call expression, wherever it occurs, rather than only as a complete statement. Second, in the spirit of the traditional patch syntax, Coccinelle allows adding and removing any sequence of tokens, as long as the pattern to match against the existing code and the pattern to create the generated code each represents a well-formed term in the C language (statement, expression, type, etc.). Thus, rather than removing the entire `end_request` call, we can leave the existing code in place and simply add a new first argument. A semantic patch incorporating both variations is as follows:

```
@@ expression res; @@
 end_request(
+  CURRENT,
   res)
```

In either case, there is no danger of matching a call to `swimiop_send_request`, as Coccinelle tokenizes and parses the code according to the grammar of the C language.

4.2 Changes in the usage of `usb_register_dev` and `usb_deregister_dev`

To make the change for `usb_register_dev` (Figure 2), it is necessary to find the values of certain members of the structure that is the first argument of the call to `usb_register_dev`. Coccinelle processes each top-level declaration (function, variable, type, etc.) separately, so matching both the structure definition and the call to `usb_register_dev` requires multiple rules. We construct a semantic patch that first matches the structure and then uses the collected information to update any call to `usb_register_dev` that refers to that structure. While most SmPL patterns, as illustrated in Section 4.1, only match code that has the exact form presented, there is an exception in the case of structure initializations. As the C language only requires specifying values for the non-zero members and allows the member values to be specified in any order, the first rule, below, matches any structure initialization that provides values for at least the members `fops` and `minor`, in any order. We give this first semantic patch rule the name `rule1` (line 1), so that the metavariable bindings that it creates can be referred to in subsequent rules.

```
1 @ rule1 @
2 identifier I;
3 expression fops_val, minor_val;
4 @@
```

```
struct usb_driver I = {
        fops:           fops_val,
        minor:          minor_val,
};
```

Equipped with the information stored in the `fops` and `minor` members of the `usb_driver` structure, we then create a second rule to update the call to `usb_register_dev`. This rule *inherits* the various metavariables bound by `rule1` (lines 2 and 3). If a file initializes multiple `usb_driver` structures, this rule will be applied once per distinct set of bindings for the inherited metavariables.

```
1 @@
2 identifier rule1.I;
3 expression rule1.fops_val, rule1.minor_val, E1, E2;
4 @@
5    usb_register_dev(
6 -     &I
7 +     fops_val, minor_val
8       , E1, E2)
```

The transformation for `usb_deregister_dev`, shown below, is then straightforward. Note that the rule could have been written to simply remove the first argument, regardless of its form. However, the following rule checks that this argument is the same as the name of the matched structure. In this way, the transformation will only be carried out if it can be carried out for both the `usb_register_dev` and `usb_deregister_dev` calls. If the needed information is not available, both calls will remain with the wrong number of arguments, having the wrong type. Untransformed code will thus cause an error when compiled, signaling to the user that the semantic patch has to be extended to take some other conditions into account.

```
@@
expression E1, E2;
identifier rule1.I;
@@
  usb_deregister_dev(
-                    &I,
                     E1, E2);
```

4.3 Introduction of `kzalloc`

To introduce the use of `kzalloc`, it is necessary to find a call to `kmalloc` such that the entire allocated region is subsequently zeroed using `memset`. As illustrated by the example patch in Figure 4, the calls are not necessarily adjacent, and indeed they may be separated by some code that is specific to the local context. Instead, we need to ensure that every execution that passes through the call to `kmalloc` also includes execution of a call to `memset`. To describe such control-flow paths, Coccinelle provides the pattern element "`...`".

A simple semantic patch introducing `kzalloc` is shown below:

```
@@
expression x, size, flag;
@@
- x = kmalloc(size, flag);
+ x = kzalloc(size, flag);
  ...
- memset(x, 0, size);
```

This semantic patch ensures that the return value of `kmalloc` is stored in an expression that has the same form as the expression in the first argument of `memset`, and that the second argument of `kmalloc` is an expression that has the same form as the expression in the third argument of `memset`. It also ensures, via the "`...`", that every execution path through the call to `kmalloc` that does not end up in a failure case, e.g., when `x` is detected to be NULL, leads to the call to `memset`. Coccinelle marks failure cases at parse time, by searching for `if` statements with only one branch, where the branch ends in a `return` or `goto`, which is a typical coding practice in the Linux kernel. The use of "`...`" furthermore ensures that the matched code does not contain another identical call to `kmalloc` or to `memset`, as "`...`" matches the shortest possible execution path between two terms matching the provided patterns.

On the other hand, the above semantic patch does not ensure full semantic correctness, e.g., that `x` or `size` is not redefined between the call to `kmalloc` and the call to `memset`. To check for unwanted code within an execution path, Coccinelle allows "`...`" to be annotated with `when` clauses indicating patterns of code that should not occur in the matched region. Furthermore, the above semantic patch does not accommodate the case where the size of the allocated region is expressed in different ways in the calls to `kmalloc` and `memset`, e.g., using the size of the type of `x` in the `kmalloc` call and the size of `*x` in the `memset` call. To address this issue, more rules can be used. These features are illustrated in a recent paper presenting some uses of Coccinelle in more depth [15].

Overall, Coccinelle aims for a WYSIWYG approach (or WYSIWIB – "what you see is where it bugs", in the case of semantic patches that are designed to find and possibly fix bugs [16]). A developer can write a semantic patch that corresponds to the developer's understanding of the code and the issues that should be taken into account. The developer can then test the resulting semantic patch on all or part of the Linux kernel, examine the results obtained, and extend the semantic patch if needed, to make it more defensive in the case of false positives, and to take into account more cases if some variations seem to be overlooked. Coccinelle provides very minimal syntactic safety, e.g., ensuring that an expression is replaced by another expression, but otherwise the developer can iteratively work towards a semantic patch that is as safe as necessary. Through this process, the semantic patch remains readable due to the use of patterns of C code that are laid out in a way that reflects how they appear in the program source code.

5 Implementation

The primary focus of this paper is on the user experience with Stratego/XT and with Coccinelle. Nevertheless, to achieve a user experience, it is necessary to actually implement the transformation system.

Coccinelle is implemented in OCaml. It relies on two parsers, one for SmPL and one for the C language. Both parsers were written from scratch, using the `yacc`-like parser generators `menhir` [25] and `ocamlyacc`, respectively. Creating these parsers from scratch allowed collecting exactly the information required by Coccinelle, including information about comments and whitespace; such information is traditionally discarded by compilers, but is essential to generate transformed source code that is identical to the original source code except at the places where transformations were performed. Creating a parser from scratch also allowed treating the exact variant of the C language used by the Linux kernel, made it possible to avoid macro expansion by parsing the macro uses directly, which both offers efficiency and makes it possible to match and transform macro uses [14], and enabled creating a parser for SmPL that can accept arbitrary code fragments, notably expressions, that are not designed to occur at the top level in the C grammar. More details about the parser design are presented by Padioleau [21]. The main components of the rest of the implementation of Coccinelle include an encoding of SmPL in a variant of computational tree logic (CTL) [3], inspired by the work of Lacey et al. on proving the correctness of compiler optimizations using temporal logic [12], and a custom pretty printer that reconstructs the program from the original tokens, modified by the specified transformations. In practice, it is the pretty printer that has required the greatest maintenance effort over the years.

After completing Stratego/XT, Visser designed the language workbench, Spoofax [10], with the goal of facilitating the design and implementation of domain-specific languages (DSLs), including generation of syntactic and semantic verifications and generation of an associated integrated development environment (IDE). As Coccinelle is essentially a domain-specific language for transforming C code, we may consider whether a tool such as Spoofax would have facilitated its development. While Spoofax includes many impressive features, it turns out to be a poor match for Coccinelle. A main goal of Coccinelle is to fit with the habits of the Linux developer community. It is traditional in this community to use command line tools and to not impose any particular graphical user interfaces for code development and maintenance. To fit with these traditions, Coccinelle was specifically designed as a command-line tool based on text files, that users were free to create however they wanted. In terms of parsing, most of the challenge, whether for the SmPL parser or for the C parser, is to parse C code, and to do this without expanding macros. The C language is not context free, and the goal of not expanding macros adds further complexity. The parser design proposed by Padioleau to cope with this challenge [21] involves arbitrary lookahead between the lexer and the parser, and re-encoding the tokens according to the collected information. This kind of flexibility is not offered by Spoofax, and is probably not necessary for the kind of well designed DSL that Spoofax targets. Furthermore, the type inference and name resolution offered by Spoofax are of limited use for Coccinelle, as SmPL does not offer nested scopes.

Still, existing language infrastructure can be useful in implementing variants of Coccinelle for some languages. Recently, we have started developing a variant of Coccinelle targeting Rust. For this, we are reusing the parser and abstract-syntax tree offered by Rust Analyzer [6]. Like C, Rust offers macros, but macro uses must conform to the syntax of the Rust language. Rust Analyzer, unlike the off-the-shelf parsers existing for C at the time when we developed Coccinelle, also retains all whitespace and comments. While Coccinelle for Rust remains work in progress, we hope that the use of Rust Analyzer will lead to a simpler implementation.

6 Conclusion

Coccinelle has been under development for more than 15 years. It is used extensively in the Linux kernel and in other C software. It continues to evolve, to better support the C code as found in the Linux kernel (a recent request was for better parsing of Sparse attributes [2]), and to support similar programming languages, such as C++. At the heart of the Coccinelle philosophy is the goal of doing one thing and doing it well, in a tool that is usable in practice by domain experts. A similar spirit also infused Eelco Visser's work.

References

1 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

2 Neil Brown. Sparse: a look under the hood. *Linux Weekly News*, 2016. `https://lwn.net/Articles/689907/`.

3 Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL*, pages 114–126, 2009.

4 Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

5 Yvonne Coady, Gregor Kiczales, Michael J. Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In A Min Tjoa and Volker Gruhn, editors, *ESEC/FSE*, pages 88–98, 2001.

6 Ferrous Systems & contributors. rust.analyzer, 2023. `https://rust-analyzer.github.io/`.

7 Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–16, 2000.

8 Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

9 Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2002.

10 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Object oriented programming systems languages and applications (OOPSLA)*, pages 444–463, October 201.

11 Ted Kremenek, Paul Twohey, Godmar Back, Andrew Y. Ng, and Dawson R. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, pages 161–176, 2006.

12 David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Principles of Programming Languages (POPL)*, pages 283–294. ACM, 2002.

13 Julia Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. Finding error handling bugs in OpenSSL using Coccinelle. In *EDCC*, pages 191–196, 2010.

14 Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX ATC*, pages 601–614, 2018.

15 Julia Lawall and Gilles Muller. Automating program transformation with Coccinelle. In *NASA Formal Methods (invited talk)*, volume 13260 of *Lecture Notes in Computer Science*, pages 71–87, 2022.

16 Julia L. Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process. *Software: Practice and Experience*, 43(1):67–92, 2013.

17 David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch.* Network Theory Ltd, January 2003.

18 Michele Martone and Julia Lawall. Refactoring for performance with semantic patching: Case study with recipes. In *High Performance Computing - ISC High Performance Digital 2021 International Workshops*, volume 12761 of *Lecture Notes in Computer Science*, pages 226–232, 2021.

19 Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *OSDI*, pages 17–30, 2000.

20 George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, 2002.

21 Yoann Padioleau. Parsing C/C++ code without pre-processing. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction (CC)*, volume 5501 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2009.

22 Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

23 Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys*, pages 59–71, 2006.

24 Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. Faults in Linux 2.6. *ACM Transactions on Computer Systems*, 32(2):4:1–4:40, 2014.

25 François Pottier and Yann Régis-Gianas. Menhir reference manual, February 2022. `http://gallium.inria.fr/~fpottier/menhir/`.

26 Laurent Réveillère and Gilles Muller. Improving driver robustness: An evaluation of the Devil approach. In *DSN*, pages 131–140, 2001.

27 Luis R. Rodriguez and Julia Lawall. Increasing automation in the backporting of Linux drivers using Coccinelle. In *EDCC*, pages 132–143, 2015.

28 Mark van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.

29 Arie van Deursen, Jan Heering, and Paul Klint. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

30 Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

31 Eelco Visser and Zine-El-Abidine Benaissa. A core language for rewriting. In *International Workshop on Rewriting Logic and its Applications, (WRLA)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 422–441. Elsevier, 1998.

32 Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.

33 Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.