A delta-oriented approach to support the safe reuse of black-box code rewriters

Benjamin Benni, Sébastien Mosser, Naouel Moha, Michel Riveill

Received XX XXXX XXXX; | Revised XX XXXX XXXX; | Accepted XX XXXX XXXX
DOI: xxx/xxxx

ARTICLE TYPE

A Delta-oriented Approach to Support the Safe Reuse of Black-box Code Rewriters

Benjamin Benni*[1] | Sébastien Mosser[2] | Naouel Moha[2] | Michel Riveill[1]

[1]Université Côte d'Azur, CNRS, I3S, Nice, France
[2] Université du Québec à Montréal, Montréal, Québec, Canada

Correspondence
*Benjamin Benni Email: benni@i3s.unice.fr

Summary

Large-scale corrective and perfective maintenance is often automated thanks to rewriting rules using tools such as `Python2to3`, `Spoon` or `Coccinelle`. Such tools consider these rules as black-boxes and compose multiple rules by chaining them: giving the output of a given rewriting rule as input to the next one. It is up to the developer to identify the right order (if it exists) among all the different rules to yield the right program. In this paper, we define a formal model compatible with the black-box assumption that reifies the modifications (Δs) made by each rule. Leveraging these Δs, we propose a way to safely compose multiple rules when applied to the same program by *(i)* ensuring the isolated application of the different rules and *(ii)* identifying unexpected behaviors that were silently ignored before. We assess this approach on two large-scale case studies: *(i)* identifying conflicts in the Linux source-code automated maintenance and *(ii)* fixing energy anti-patterns existing in Android applications available on GitHub.

KEYWORDS:
Code Rewriting, Software Reuse, Conflict detection, Rule Composition

1 | INTRODUCTION

It is commonplace to state that "software evolves", and it is part of software developers' duty to support and operate such an evolution. The adaptive, preventive, corrective, and perfective evolution [1,2] of a piece of software to address new requirements is taken into account by software development methodologies and project management methods [3]. But this evolution is not related to the addition of immediate business value in the program. It covers time-consuming and error prone activities, including software migration (*e.g.*, moving from `Python 2.x` to `Python 3.x`), framework upgrade (*e.g.*, supporting upcoming versions of `Android` for a mobile application), implementation of best practices that change along time (*e.g.*, following vendor guidelines to rewrite a deployment descriptors), code refactoring (*e.g.*, to introduce design patterns), or bugs/anti-patterns correction.

It is usual to automate the evolution as much as possible, using tools working directly on the source code. Thus, one will *reuse* such automation to either maintain the software (*e.g.*, bug fixing, syntactical guideline conformance) or make it evolve (*e.g.*, API changes). For example, migrating from `Python 2.x` to `3.x` can be automated using the `2to3` shell command [4]. This tool provides 50 *fixer*s, each one implementing a specific evolution made to the Python *Application Programming Interface* (`API`), *e.g.*, iterating over a collection, modification made to the `String API`, idiomatic patterns to be used in the latest version. One can (un-)select the evolutions (*i.e.*, the fixers) she wants to apply to her program and run the tool to obtain a Python 3 compatible version of an initial Python 2 program with respect to the available rules. In a broader way, any up-to-date `IDE` provides automated refactoring options to ease the work of software developers.

In 2006, Muller *et al.* coined the term of *collateral evolution* to address the issues that appear when developing `Linux` drivers: the kernel libraries continuously evolve, and device-specific drivers must be ported to support the new `API`s. To tame this challenge, they develop the `Coccinelle`

tool[5], used to rewrite C code and generate patches automatically applied to the Linux kernel to correct bugs or adapt drivers, in a fully automated way. In the Java ecosystem, the Spoon tool[6] (also released in 2006) allows one to write processors that adapt Java source code in various way, such as code refactoring, automated bug-fixing or anti-pattern fixing[7]. At runtime, these three tools (2to3, Spoon, and Coccinelle) consider rewriting rules as black-boxes, applied to a program to generate a patched one. These tools are what we call *Code rewriters*: they modify a codebase for a given purpose, given rewriting rules.

Contrarily to applications (*e.g.*, *abstract rewriting machines*) that focus on the confluence, fixed point identification and termination of the rewriting process for a given set of rewriting rules[8], the previously cited tools take an opposite point of view. They do not consider the confluence of rule applications, and generalize the classical function composition operator ($\circ$) to compose rules: each rule r_i is a black-box that consumes a program p to produce a new program p′. In the example shown above, rules are *sequentially applied* to the input program to yield the final one, passing intermediate results to each other. There are no additional processing applied after these steps that might consider the set of rules as a first class entity.

$$p' = apply(p, [r_1, \ldots, r_n]) = r_1 \circ \cdots \circ r_n(p) = r_1(\ldots(r_n(p)))$$

The main issue with this assumption is the impact of overlapping rules on the yield program. Considering large software systems where separation of concerns matters, each code rewriting function is defined independently. As a consequence, if two rules do not commute ($r_1(r_2(p)) \neq r_2(r_1(p))$), it is up to the developer to *(i)* identify that *these* rules are conflicting inside the whole rule sequence, and *(ii)* fix the rules or the application sequence to yield the expected result.

To mitigates these issues, different approaches exist in the literature, working on different software aspects. Merge techniques exist that either require a white-box approach (*e.g.*, by building a graph of a software with function definitions and invocations)[9], or are focused on a very specific domain[10]. Behavioral conflict detections rely on complex mathematical formalisms that need *domain-specific* tooling, making them not reusable. Classical rewriting methods (*e.g.*, term-rewriting and graph-rewriting) cannot be applied to the tools presented above, as they break the underlying black-box assumption: in our context, a rewriting function cannot be opened to reason on its intrinsic definition, and only the result of its application on a given program can be analyzed. *Even if* such white-box approach *would* be applicable, analyzing and formalizing rewriting functions would either need too much effort or would be impossible to do automatically. For instance, how can one reason on an arbitrary source code, without convention nor asumption on its content, goal, language, or target, or even formalizing it? One of the objectives of this work is to be compatible with state of the art tools used to automatically rewrite source code in an industrial context. The tools under consideration in our study (i.e., 2to3, Coccinelle, Spoon) do not expose pre-conditions, and even if Coccinelle might be compatible with a static analysis thanks to its "pattern matching" approach to automatically infer such preconditions, the two other tools are using regular code that make such an inference extremely hazardous. Moreover, Spoon code in Java often relies on reflexive construction, making its static analysis impossible. To the best of our knowledge, it is not possible to consider the legacy spoon processors or coccinelle patch and automatically transform them into any graph rewriting formalism. This fact prevents any large scale validation on real code as the one performed in this submission. We therefore propose a formalism that matches the state-of-practice associated to the Linux and Java ecosystems : *(i)* an action-based representation of code modifications (supported by approaches such as Praxis or Adore), and *(ii)* a diff-based mechanism at the AST level when necessary (supported by GumTree[11] or JDime[12] tools). For more information about these related works, we kindly ask the reader to read Sec. 6.

In this paper, **we propose a domain-independent approach to support the safe reuse of code rewriters defined as black-boxes**. The originality of the approach is to reason on the modifications (*i.e.*, the *deltas*) produced by code rewriters, when their application to the program guarantees that each one is successfully applied, or to detect conflicts when relevant. We give in Sec. 2 background information about black-box rewriters, using Coccinelle and Spoon as examples. Then, Sec. 3 defines a formal model to represent deltas and Sec. 4 describes how conflicts can be identified based on this representation. The approach is applied on two different real-life case studies: Linux source-code and Android applications. Sec. 5 describes how the contribution is implemented and then applied to identify rewriting conflicts when automatically patching *(i)* 19 different versions of Linux source-code (each representing 20M lines of code) with respect to 35 maintainer guidelines, and *(ii)* 22 Android mobile applications with respect to Google guidelines for energy efficiency. All things considered, we estimate that the work done to perform such a large scale validation involved 4.5 person-months at MSc and PhD level, plus supervision, this includes both Linux experimental setup and Android experimental set-up; and the definition and development of tools to identify rewriting rule conflicts in the Android ecosystem. Finally, Sec. 6 describes related work and Sec. 7 concludes the paper by describing perspectives of this work from both theoretical and empirical point of views.

2 | BACKGROUND AND CHALLENGES

In this section, we focus on the two tools that exist in the state of practice in the Linux and Android actual eco-systems (Coccinelle and Spoon) to automate code rewriting, so to identify the challenges our contribution addresses.

```
1 @@
2 type T;
3 expression x, E, E1,E2;
4 @@
5 - x = kmalloc(E1,E2);
6 + x = kzalloc(E1,E2);
7 ... when != \( x[...]=E; \| x=E; \)
8 - memset((T) x, 0, E1);
```

(a) `kmalloc` ∧ `memset`(0) ↦ `kzalloc` (R_k)

```
1 @@
2 type T;
3 T *x;
4 expression E;
5 @@
6
7 - memset(x, E, sizeof(x))
8 + memset(x, E, sizeof(*x))
```

(b) Fix size in `memset` call (R_m)

```
1 struct Point {
2   double x;
3   double y;
4 };
5 typedef struct Point Point;
6
7 int main()
8 {
9   Point *a;
10  // ....
11  a = kmalloc(sizeof(*a), 0);
12  // not using a
13  memset(a, 0, sizeof(a));
14  // ...
15  return 0;
16 }
```

(c) Example of a C program (p_c)

```
1 // Expected Program: Pkm = Rk(Rm(Pc))
2 int main()
3 {
4   Point *a;
5   // ....
6   a = kzalloc(sizeof(*a), 0);
7   // ...
8   return 0;
9 }
10
11 // Incompletely fixed program: Pmk = Rm(Rk(Pc))
12 int main()
13 {
14   Point *a;
15   // ....
16   a = kmalloc(sizeof(*a), 0);
17   // not using a
18   memset(a, 0, sizeof(*a));
19   // ...
20   return 0;
21 }
```

(d) Applying rules R_k and R_m to p_c.

FIGURE 1 Coccinelle: using semantic patches to rewrite C code

2.1 | Using Coccinelle to patch the Linux Kernel

As stated in the introduction, the `Coccinelle` tool is used to automatically fix bugs in the `C` code that implements the `Linux` kernel, as well as backporting device-specific drivers [13]. These activities are supported by allowing a software developer to define *semantic patches*. A semantic patch contains *(i)* the declaration of free variables in a header identified by *at* symbols (`@@`), and *(ii)* the patterns to be matched in the `C` code coupled to the rewriting rule. Statements to remove from the code are prefixed by a minus symbol (–), statements to be added are prefixed by a plus symbol (+), and placeholders use the "..." wildcard. For example, the rule R_k (Fig. 1a) illustrates how to rewrite legacy code in order to use a new function available in the kernel library instead of the previous API. It describes a semantic patch removing any call to the kernel memory-allocation function (`kmalloc`, *line 5*) that is initialized with 0 values (`memset`, *line 8*), and replacing it by an atomic call to `kzalloc` (*line 6*), doing allocation and initialization at the same time. Wildcard patterns can define guards, for example here the patch cannot be applied if the allocated memory was changed in between (using the `when` keyword, line7). Fig. 1b describes another semantic patch used to fix a very common bug, where the memory initialization is not done properly when using pointers (*line 8*). These two examples are excerpts of the examples available on the tool webpage[1].

Considering these two semantic patches, the intention of applying the first one (R_k) is to use a call to `kzalloc` whenever possible in the source code, and the intention associated to the second one (R_m) is to fix a bad memory allocation. In the state of practice, applying the two patches, in any order, does not yield any error. However, the application order matters. For example, when applied to the sample program p_c described in Fig. 1c:

- $p_{km} = R_k(R_m(p_c))$: The erroneous `memset` is fixed (Fig. 1c, *line 13*), and as a consequence the `kzalloc` optimization is also applied to the fixed `memset`, merging *line 11* and *line 13* into a single memory allocation call. In this order, the two initial intentions are respected in p_{km}: all the erroneous memory allocations are fixed, and the atomic function `kzalloc` is called whenever possible. This is the expected result depicted in the upper part of Fig. 1d.
- $p_{mk} = R_m(R_k(p_c))$: In this order, the erroneous memory allocations are fixed after the `kzalloc` merge. As a consequence, it is possible to miss some of these `kzalloc` calls when it implies badly defined `memset`. Considering p_c, *line 11* and *line 13* are not mergeable until *line*

[1] http://coccinelle.lip6.fr/impact_linux.php. The "kzalloc treewide" semantic patch implements R_k and the "fix size given to memset" one implements R_m.

```
1 public class NPGuard extends AbstractProcessor<CtClass> {
2
3   @Override public boolean isToBeProcessed(CtClass candidate) {
4     List<CtMethod> allMethods = getAllMethods(candidate);
5     settersToModify = keepSetters(allMethods);
6     return !settersToModify.isEmpty();
7   }
8
9   @Override public void process(CtClass ctClass) {
10    List<CtMethod> setters = settersToModify;
11    for (CtExecutable currentSetterMethod : setters) {
12      if (isASetter(currentSetterMethod)) {
13        CtParameter parameter =
14          (CtParameter) currentSetterMethod.getParameters().get(0);
15        CtIf ctIf = getFactory().createIf();
16        ctIf.setThenStatement(currentSetterMethod.getBody().clone());
17        String snippet = parameter.getSimpleName() + " != null";
18        ctIf.setCondition(getFactory()
19          .createCodeSnippetExpression(snippet);
20        currentSetterMethod.setBody(ctIf);
21      }
22    }
23  }
24 }
```

FIGURE 2 Spoon: using processors to rewrite Java code (`NPGuard.java`, R_{np}))

11 pointer is fixed, leading to a program p_{mk} where the intention of R_k is not respected: the `kzalloc` method is not called whenever it is possible in the final program (lower part of Fig. 1d).

2.2 | Using Spoon to fix anti-patterns in Android applications

`Spoon` is a tool defined on top of the `Java` language, which works at the *Abstract Syntax Tree* (AST) level. It provides the AST of a `Java` source code and lets the developer define her transformations. A `Spoon` rewriter is modeled as a `Processor`, which implements an AST to AST transformation. It is a `Java` class that analyses an AST by filtering portions of it (identified by a method named `isToBeProcessed`), and applies a `process` method to each filtered element, modifying this AST. `Spoon` reifies the AST through a meta-model where all classes are prefixed by `Ct`: a `CtClass` contains a set of `CtMethod` made of `CtExpressions`. An example of processor is given in Fig. 2.

We consider here two processors defined according to two different intentions. The first one, implemented in a file `NPGuard.java` (R_{np}), is a rewriter used to protect setters[2] from null pointer assignment by introducing a test that prevents an assignment to the *null* value to an instance variable in a class (Fig. 2). The second one (`IGSInliner.java`, R_{igs}) implements a guideline provided by `Google` when developing mobile applications in `Java` using the `Android` framework. Inside a given class, a developer should directly use an instance variable instead of accessing it through its own getter or setter (*Internal Getters Setters* anti-pattern). This is one (among many) way to improve the energy efficiency of the developed application[14] with `Android`[3].

Like in the `Coccinelle` example, these two processors work well when applied to `Java` code, and always yield a result. However, order matters as there is an overlap between the addition of the *null* check in R_{np} and the inlining process implemented by R_{igs}. As described in Fig. 3, when composing these two rules, if the guard mechanism is introduced before the setters are inlined, the setters will *not* be inlined as they do not conform to the setter definition with the newly introduced `if` statement. We depict in Fig. 3 how these processors behave on a simple class p_j. Inlining setters yields p_{igs}, where internal calls to the `setData` method are replaced by the contents of the associated method (Fig. 3b, *line 11*). When introducing the *null* guard, the contents of the `setData` method is changed (Fig. 3c, *lines 5-8*), which prevents any upcoming inlining as `setData` is not considered as a setter based on its new implementation and the used definition. As a consequence, $R_{igs}(R_{np}(p_j)) = R_{np}(p_j)$, and R_{igs} is useless in this very context. It is interesting to remark that, when considering R_{igs} and R_{np} to be applied to the very same program, one actually expects the result described in Fig. 3d: internal setters are inlined with the initial contents of `setData`, and any external call to `setData` is protected by the guard.

[2]We use the classical definition of a setter, *i.e.*, *"a setter for a private attribute `x` is a method named `setX`, with a single parameter, and doing a single-line and type-compatible assignment from its parameter to `x`"*.

[3]http://stackoverflow.com/a/4930538

```java
public class C {

  private String data;

  public String setData(String s) {
    this.data = s;
  }

  public void doSomething() {
    // ...
    setData(newValue)    /* <<<< */
    // ...
  }
}
```

(a) Example of a Java class (`C.java`, p_j)

```java
public class C {

  private String data;

  public String setData(String s) {
    this.data = s;
  }

  public void doSomething() {
    // ...
    this.data = newValue    /* <<<< */
    // ...
  }
}
```

(b) $p_{igs} = R_{igs}(p_j)$

```java
public class C {

  private String data;

  public String setData(String s) {
    if (s != null)
      this.data = s;
  }

  public void doSomething() {
    // ...
    setData(newValue)     /* <<<< */
    // ...
  }
}
```

(c) $p_{np} = p_{igs \circ np} = R_{igs}(R_{np}(p_j))$

```java
public class C {

  private String data;

  public String setData(String s) {
    if (s != null)
      this.data = s;
  }

  public void doSomething() {
    // ...
    this.data = newValue    /* <<<< */
    // ...
  }
}
```

(d) $p_{np \circ igs} = R_{np}(R_{igs}(p_j))$

FIGURE 3 Spoon: applying processors to Java code

2.3 | Challenges associated to rewriting rules reuse

Based on these two examples that come from very different worlds, we identify the following challenges that need to be addressed to properly support the safe reuse of code rewriters. These challenges define the scope of requirements associated to our contribution. As rewriting tools are part of the state of practice in software engineering (*e.g.*, for scalability purpose when patching the whole `Linux` kernel), an approach supporting the reuse of rewriting rules must be aligned with the assumptions made by these tools, *i.e.*, considering their internal decisions as black-boxes.

- C_1 *Rule isolation*. When rules overlap, it is not possible to apply the two rules in isolation, as the result of a rule is used to feed the other one. It is important to support isolation when possible, avoiding irrelevant conflicts only introduced by the *sequential application* of the rules.
- C_2 *Conflict detection*. As each rewriter is associated to an *intention* (e.g., removing memory allocation that does not use a specific API), it is important to provide a way to assess if the initial intention is still valid in the composed result.

3 | USING DELTAS TO ISOLATE RULE APPLICATIONS (C_1)

In this section, we focus on the definition of a formal model that supports the safe reuse of code rewriters, *w.r.t.* the challenges identified in the previous section. This model directly addresses the first challenge of *rule isolation* (C_1). It also provides elementary bricks to support the *conflict detection* one (C_2, Sec. 4).

We model a code rewriter $\rho \in P$ as a pair of two elements: *(i)* a function $\varphi \in \Phi$ used to rewrite the AST, coupled to *(ii)* a checker function $\chi \in X$ used to validate a postcondition associated to the rewriting. The postcondition validation is modeled as a boolean function taking as input the initial AST and the resulting one, returning *true* when the postcondition is valid, and false otherwise. For a given $p \in AST$, applying φ to it yields an AST p' where $\chi(p, p')$ holds. For instance, the rewriter φ_b that removes the useless internal setter calls, described in Sec.2.2 and for which the result is depicted in Fig.3b ; comes with a post-condition χ_b that states that "the resulting code does not contain calls to internal setter method".

As any programming activity, it is up to the developer to properly implement her intention in the post-condition definition. This is more easy in an expressive language, even if tricky to debug. In situations where the rewriting language cannot support easily such a definition, it is possible to approximate the postcondition by reusing the rewriting function and assessing if it does not yield any new modification to be applied on the source code. It implies the definition of idempotent rules, but this is relevant in an industrial context as for example the Linux ecosystem support such an assumption. This is the design choice we made for modelling postconditions: if applying a rule once produces modifications ; applying it twice should not yield any modifications.

Contrarily to classical graph-rewriting approaches, we do not formalize preconditions as the state-of-practice-tools silently return the given AST when they are not applicable without providing any concrete precondition check. The tools under consideration in our study (*i.e.*, 2to3, Coccinelle, Spoon) do not expose pre-conditions, meaning that the preconditions are not known or even computable. For example, considering how Spoon code is classicaly used, it intensively relies on the reflection API available in Java, making static analysis extremely difficult in real life programs.

This model supports the formalization of the rewriting rules exemplified in the previous section, and also automates the validation of the developer's intention on the yield program.

$$\begin{gathered}\text{Let } \rho = (\varphi, \chi) \in (\Phi \times \mathsf{X}) = \mathsf{P}, \quad (\varphi : \mathsf{AST} \to \mathsf{AST}) \in \Phi \\ \chi : \mathsf{AST} \times \mathsf{AST} \to \mathbb{B} \in \mathsf{X}, \qquad \forall \mathsf{p} \in \mathsf{AST},\ \chi(\mathsf{p}, \varphi(\mathsf{p})) \\ \varphi(\mathsf{p}) = \mathsf{p} \text{ if } \varphi \text{ is not applicable to } \mathsf{p}\end{gathered} \tag{1}$$

Working with functions that operate at the AST level does not provide any support to compose these functions excepting the classical composition operator $\circ$. When combined with the postcondition validation introduced in the model, it supports the apply operator that classically exists in the rewriting tools. The rules $[\rho_1, \ldots, \rho_n]$ to be applied are consumed in *sequence* (*i.e.*, $\mathsf{P}^n_<$ is an ordered sequence of rules), leading to a situation where only the last postcondition (χ_1) can be ensured in the resulting program, by construction. This rule composition operator, named apply, models the classical behavior for rule composition in the state of practice. Note that since $\circ$ is a sequential operator, it unfolds into $\varphi_1(\varphi_2(\ldots(\varphi_n(\mathsf{p}))))$. Thus, only the last postcondition *to be applied*, *i.e.*, χ_1, holds.

$$\begin{gathered}\mathsf{apply} : \mathsf{AST} \times \mathsf{P}^n_< \to \mathsf{AST} \\ \mathsf{p}, [\rho_1, \ldots, \rho_n] \mapsto \text{Let } \mathsf{p}_{2.n} = (\overset{n}{\underset{i=2}{\circ}} \varphi_i)(\mathsf{p}),\ \mathsf{p}' = \varphi_1(\mathsf{p}_{2.n}), \quad \chi_1(\mathsf{p}_{2.n}, \mathsf{p}')\end{gathered} \tag{2}$$

As shown in the previous section, using this operator leads to scheduling issues, as it implies strong assumptions on the functions being commutative. We build here our contribution on top of two research results: Praxis [15] and a parallel composition operator [16]. These two approaches share in common the fact that instead of working on a model (here an AST), they rely on the sequence of elementary actions used to build it. Praxis demonstrated that for any model m, there exists an ordered sequence of elementary actions $[\alpha_1, \ldots, \alpha_n] \in \mathsf{A}^*_<$ yielding m when applied to the empty model. The four kinds of actions available in A are:

1. the creation of a model element (`create`),
2. the deletion of a model element (`delete`),
3. setting a property to a given value in a model element (`setProp`),
4. and setting a reference that binds together two model elements (`setRef`).

For example, to build the `Java` program $\mathsf{p_j}$ described in Fig. 3a, one can use a sequence of actions $\mathsf{S_{p_j}}$ that creates a class, names it `C`, creates an instance variable, names it `data`, sets its type to `String`, adds it to `C`, *etc*.

$$S_{p_j} = [create(e_1, Class), setProperty(e_1, name, \{\text{“}C\text{”}\}), create(e_2, Field), \ldots, add(e_2, e_1, \text{“}fields\text{”}), \ldots] \in A^*_<$$

Considering a rewriting rule as an action-producer introduces the notion of *deltas* in the formalism. We consider the rewriting not through its resulting AST but through the elementary modifications made on the AST (*i.e.*, a Δ) by the rule to produce the new one. This is compatible with the “rules are black boxes" assumption for two reasons. On the one hand, rewriting tools use a similar mechanism in their rewriting engine. We model here the application of a sequence of actions to a given AST to modify it, as a generic operator denoted by $\oplus$. This operator applies an ordered sequence of actions to an AST but a concrete application of a given action on the AST is a language-specific operation. For example the $\oplus$ operator can rely on patches for `Coccinelle` (*i.e.*, Δs containing elements to add or remove) or on an action model for `Spoon` (*e.g.*, *line 15* in Fig. 2 creates an `if` statement, and *line 20* binds the contents of the setter to this new conditional statement).

On the other hand, it is possible for certain languages to define a differencing tool working at the AST level. For example, the `GumTree` tool [11] exposes the differences between two `Java` ASTs as the minimal sequence of actions necessary to go from one AST to the other one. We denote

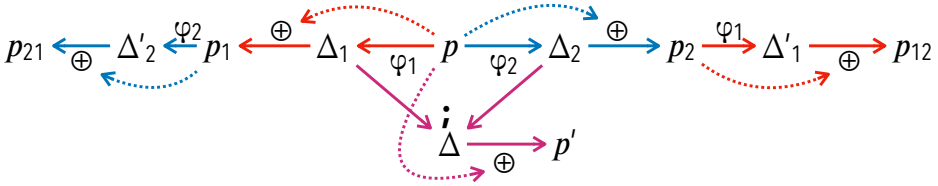


FIGURE 4 Sequential ($p \mapsto \{p_{12}, p_{21}\}$) *versus* isolated ($p \mapsto p'$) rewriting

such a *diff* operation using the $\ominus$ symbol (language-specific).

$$\begin{aligned} &\oplus : \mathsf{AST} \times \mathsf{A}^*_< \to \mathsf{AST} \\ &\qquad (p, S) \mapsto \begin{cases} S = \emptyset & \Rightarrow p \\ S = \alpha | S' & \Rightarrow \mathsf{exec}(\alpha, p) \oplus S', \ \mathsf{exec}\ \text{being language specific.} \end{cases} \\ &\ominus : \mathsf{AST} \times \mathsf{AST} \to \mathsf{A}^*_< \\ &\qquad (p', p) \mapsto \Delta, \ \text{where } p' = p \oplus \Delta \end{aligned} \tag{3}$$

This representation is compatible with the previously defined semantics for the apply composition operator.

$$\begin{aligned} &\text{Let } p \in \mathsf{AST},\ \rho_1 = (\varphi_1, \chi_1) \in \mathsf{P}, \rho_2 = (\varphi_2, \chi_2) \in \mathsf{P} \\ &\quad p_1 = \varphi_1(p) = p \oplus (p_1 \ominus p) = p \oplus \Delta_1, && \chi_1(p, p_1) \\ &\quad p_2 = \varphi_2(p) = p \oplus (p_2 \ominus p) = p \oplus \Delta_2, && \chi_2(p, p_2) \\ &\quad p_{12} = \mathsf{apply}(p, [\rho_1, \rho_2]) = \varphi_1 \circ \varphi_2(p) = \varphi_1(\varphi_2(p)) \\ &\qquad = \varphi_1(p \oplus \Delta_2) = (p \oplus \Delta_2) \oplus \Delta'_1, && \chi_1(p_2, p_{12}) \\ &\quad p_{21} = \mathsf{apply}(p, [\rho_2, \rho_1]) = \varphi_2 \circ \varphi_1(p) = \varphi_2(\varphi_1(p)) \\ &\qquad = \varphi_2(p \oplus \Delta_1) = (p \oplus \Delta_1) \oplus \Delta'_2, && \chi_2(p_1, p_{21}) \end{aligned} \tag{4}$$

A rewriting rule is formalized as an action-producer, and it is part of our hypothesis that: *(i)* during the application of a rewriting rule (*i.e.*, when applying a subset of its actions) the AST may be in a inconsistent state, but *(ii)* applying a rule (*i.e.*, all of its actions) *must* end in a consistent and correct result.

Extending the state of practice.

However, the need for the user to decide an order is implied by the way the rewriting tools are implemented. At the semantic level, the user might not want to order the different rewritings (as seen in the `Spoon` example). Using our model, it is possible to leverage the Δs to support an **isolated composition** of multiple rewriting rules, where the rewriting functions are applied on the very same model in an isolated way (Fig. 4). Using this approach, *(i)* we obtain the two sequences Δ_1 and Δ_2 used to yield p_1 and p_2, *(ii)* concatenate[4] them into a single one Δ, and *(iii)* apply the result to the initial program. As a consequence, according to this composition semantics, both postconditions χ_1 and χ_2 must hold in the resulting program p' for it to be valid. An interesting property of the isolated composition is to ensure that all postconditions are valid when applied to a program.

$$p' = p \oplus ((p_1 \ominus p); (p_2 \ominus p)) = p \oplus (\Delta_1; \Delta_2), \quad \chi_1(p, p') \wedge \chi_2(p, p') \tag{5}$$

Unfortunately, it is not always possible to apply rules in an isolated way: for example, to yield the expected program in the `Coccinelle` example (Fig. 1d), it is necessary to always execute the error fixing rule before the allocation optimization one. However, we need to detect that one ordering ensures both postconditions, where the other only ensures the last one. As a consequence, we generalize the application of several rewriting rules to a given program according to two new composition operators that complement the legacy `apply` operator. Using these operators ensures that all the postconditions hold between the initial program and the final one, no matter what happened in between.

Thus, the `seq` operator implements the sequential composition of an ordered sequence of rules (*i.e.*, what is actually used in the examples shown in the introduction), and the `iso` operator implements the isolated application of a set of rules. Using the `seq` operator, each rule is applied

[4]We consider a function denoted as ; that implements action sequence concatenation.

sequentially, but the process expects all the postconditions to hold at the end (where using apply, only the last postcondition holds).

$$\begin{aligned}
&\mathsf{seq} : \mathsf{AST} \times \mathsf{P}^{n}_{<} \to \mathsf{AST} \\
&\quad \mathsf{p}, [\rho_1, \ldots, \rho_n] \mapsto \mathsf{p}_{\mathsf{seq}} = (\overset{n}{\underset{i=1}{\circ}} \varphi_i)(\mathsf{p}), \qquad \overset{n}{\underset{i=1}{\wedge}} \chi_i(\mathsf{p}, \mathsf{p}_{\mathsf{seq}}) \\
&\mathsf{iso} : \mathsf{AST} \times \mathsf{P}^{n} \to \mathsf{AST} \\
&\quad \mathsf{p}, \{\rho_1, \ldots, \rho_n\} \mapsto \mathsf{p}_{\mathsf{iso}} = \mathsf{p} \oplus (\overset{n}{\underset{i=1}{;}} (\varphi_i(\mathsf{p}) \ominus \mathsf{p})), \quad \overset{n}{\underset{i=1}{\wedge}} \chi_i(\mathsf{p}, \mathsf{p}_{\mathsf{iso}})
\end{aligned} \tag{6}$$

4 | DETECTING SYNTACTIC AND SEMANTIC CONFLICTS (C_2)

We discriminate conflicts according to two types: *(i)* syntactic conflicts and *(ii)* semantic conflicts. The latter are related to the violation of postconditions associated to the rewriting rules. The former are a side effect of the iso operator, considering that Δs might perform concurrent modifications of the very same tree elements when applied in an isolated way. These two mechanisms address the second challenge of *conflict detection* (C_2, Sec. 2).

4.1 | Syntactic conflicts as overlapping deltas

Let p be an AST that defines a class `C` with a *protected* attribute named `att`. Let ρ_1 and ρ_2 two rewriting rules, applied using the iso operator to prevent one to capture the output of the other. On the one hand, applying φ_1 to p creates Δ_1, which makes `att` *private*, with an associated getter and setter. On the other hand, applying φ_2 to p creates Δ_2, which promotes the very same attribute as a *public* one. As an attribute cannot be *public* and *private* at the very same time, we encounter here *a syntactic conflict*: applying the two rules ρ_1 and ρ_2 on the same program is not possible as is.

$$\begin{aligned}
&\varphi_1(\mathsf{p}) \ominus \mathsf{p} = \Delta_1 = [\ldots, setProperty(\mathsf{att}, \mathsf{visibility}, \{\text{“private”}\}), \ldots] \\
&\varphi_2(\mathsf{p}) \ominus \mathsf{p} = \Delta_2 = [\ldots, setProperty(\mathsf{att}, \mathsf{visibility}, \{\text{“public”}\}), \ldots]
\end{aligned} \tag{7}$$

On the one hand, the seq operator cannot encounter a syntactical conflict, as it is assumed to produce a valid AST as output. On the other hand, the iso operator can encounter three kinds of conflicts (Eq. 8) at the syntax level[5]: *Concurrent Property Modification* (CPM), *Concurrent Reference Modification* (CRM) and *Dangling reference* (DR). The first and second situation identify a situation where two rules set a property (or a reference) to different values. It is not possible to automatically decide which one is the right one. The last situation is identified when a rule creates a reference to a model element that is deleted by the other one. This leads to a situation where the resulting program will not compile. Thanks to the definition of these conflicting situations, it is possible to check if a pair of Δs is conflicting through the definition of a dedicated function

$$conflict? : A^*_< \times A^*_< \to \mathbb{B}$$

If this function returns *true*, it means that the two rewriting rules cannot be applied independently on the very same program. One can generalize the $conflict?$ function to a set of Δs by applying it to the elements that compose the Cartesian product of the Δs to be applied on p.

$$\begin{aligned}
\mathsf{CPM} :\ & \mathsf{A}^*_< \times \mathsf{A}^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto \exists \alpha \in \Delta, \alpha' \in \Delta', \alpha = setProperty(\mathsf{elem}, \mathsf{prop}, \mathsf{value}) \\
& \qquad \alpha' = setProperty(\mathsf{elem}, \mathsf{prop}, \mathsf{value}'), \mathsf{value} \neq \mathsf{value}' \\
\mathsf{CRM} :\ & \mathsf{A}^*_< \times \mathsf{A}^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto \exists \alpha \in \Delta, \alpha' \in \Delta', \alpha = setReference(\mathsf{elem}, ref, \mathsf{elem}') \\
& \qquad \alpha' = setReference(\mathsf{elem}, ref, \mathsf{elem}''), \mathsf{elem}' \neq \mathsf{elem}'' \\
\mathsf{DR} :\ & \mathsf{A}^*_< \times \mathsf{A}^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto \exists \alpha \in \Delta, \alpha' \in \Delta', \alpha = setReference(\mathsf{elem}, ref, \mathsf{elem}') \\
& \qquad \alpha' = delete(\mathsf{elem}''), \mathsf{elem}' = \mathsf{elem}'' \\
conflict? :\ & \mathsf{A}^*_< \times \mathsf{A}^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto \mathsf{CPM}(\Delta, \Delta') \vee \mathsf{CRM}(\Delta, \Delta') \vee \mathsf{DR}(\Delta, \Delta') \vee \mathsf{DR}(\Delta', \Delta)
\end{aligned} \tag{8}$$

[5] See the Praxis seminal paper [15] for a more comprehensive description of conflict detection in the general case.

TABLE 1 Identifying semantic conflicts on the `Coccinelle` example

$p \in AST$	$p' \in AST$	$\chi_k(p, p')$	$\chi_m(p, p')$	Postcondition
p_c	$\varphi_k(p_c)$	✓		✓
p_c	$\varphi_m(p_c)$		✓	✓
$\varphi_m(p_c)$	$apply(p_c, [\rho_k, \rho_m])$	✓	✓	✓
$\varphi_k(p_c)$	$apply(p_c, [\rho_m, \rho_k])$	×	✓	✓
p_c	$seq(p_c, [\rho_k, \rho_m])$	✓	✓	✓
p_c	$seq(p_c, [\rho_m, \rho_k])$	×	✓	×
p_c	$iso(p_c, \{\rho_k, \rho_m\})$	×	✓	×

TABLE 2 Identifying semantic conflicts on the `Spoon` example

$p \in AST$	$p' \in AST$	$\chi_{igs}(p, p')$	$\chi_{np}(p, p')$	Postcondition
p_j	$\varphi_k(p_c)$	✓		✓
p_j	$\varphi_m(p_c)$		✓	✓
$\varphi_{np}(p_j)$	$apply(p_j, [\rho_{igs}, \rho_{np}])$	✓	✓	✓
$\varphi_{igs}(p_j)$	$apply(p_j, [\rho_{np}, \rho_{igs}])$	✓	✓	✓
p_j	$seq(p_c, [\rho_{igs}, \rho_{np}])$	×	✓	×
p_j	$seq(p_c, [\rho_{np}, \rho_{igs}])$	✓	✓	✓
p_j	$iso(p_c, \{\rho_{igs}, \rho_{np}\})$	✓	✓	✓

The syntactical conflict detection gives an information to the software developer: among all the rules used to rewrite the program under consideration, there exists a pair of rules that cannot be applied independently. It is still her responsibility to fix this issue (*e.g.*, by specifiying an execution *order*, or by implementing a *specific rule* that handles this corner case), but at least the issue is explicit and scoped instead of being silently ignored.

4.2 | Semantic conflicts as postcondition violations

We now consider rewriting rules that are not conflicting at the syntactical level. We focus here on the postconditions defined for each rule, *w.r.t.* the legacy, sequential and isolated composition operators. We summarize in Tab. 1 and Tab. 2 how the different postconditions hold when applying the apply, iso and seq operators to the examples defined in Sec. 2. When composed using the apply operator ($p' = apply(p, rules)$), the only guarantee is that the last postcondition is true. It is interesting to notice that, in both tables, using the apply operator always yields a result that conforms to the associated postcondition, even if the result is not the expected one.

Let $rules = [\rho_1, \ldots, \rho_n] \in P^n$ a set of rewriting rules. When using the seq operator, ordering issues are detected. For example, in the `Coccinelle` example where one tries to fix memset calls *and* replaces malloc calls, both sequence of rules-application (*i.e.*, $apply(p_c, [\rho_k, \rho_m])$ and $apply(p_c, [\rho_m, \rho_k])$) yield a valid result: none of them violate the postconditions since the last rule-check succeeded. However, in the last case ($apply(p_c, [\rho_m, \rho_k])$), the fixed call to `memset` introduced by ρ_m makes the postcondition invalid. When using the seq operator, only $seq(p_c, [\rho_k, \rho_m])$ is valid *w.r.t.* to the postcondition associated to the operator. This detects the fact that fixing the `memset` size error must be applied before the one that merges the `kmalloc` and `memset` calls to support both intentions. The operator also identifies an issue when, in the `Spoon` example, the *guard* rule is applied before the other one.

When composed using the iso operator ($p' = iso(p, rules)$), the resulting program is valid only when all the postconditions hold when the rules are simultaneously applied to the input program. On the one hand, when applied to `Coccinelle` example, this is not the case. The fact that at least one postcondition is violated when using the iso operator gives a very important information to the developers: these two rewriting rules cannot be applied independently on this program. On the other hand, considering the `Spoon` example, the two rules can be applied in isolation (yielding the result described in Fig. 3d).

5 | VALIDATION

The goal of this section is to illustrate how the formal model described in the two previous sections is valid with respect to large-scale case studies. We present two different applications here that focus on two different dimensions: *(i)* a Linux-kernel use case that applies our proposition to a single application with a rule set that we did not define, and *(ii)* an Android use case that applies our approach on many different open-source applications given a fixed set a rules that we defined.

5.1 | Linux kernel source code

The validation material for this section can be found on the dedicated dataset available in open access [17], and has been built using the `Coccinnelle` tool, in the `Linux` ecosystem. Each rule is defined as a `Semantic patch` working at the code-level. The objective here is to show how the iso operator can be used in practice to identify conflicts among legacy rules and reduce the number of rules to order when necessary. We performed our experiments on 19 different versions of the `Linux` kernel source-code, with an Intel Xeon E5-2637v2 (3,5GHz, 8cores) processor and 64GB of RAM. We randomly picked a version per month from January 2017 to July 2018, and analysed the applications and interactions of all semantic patches available in `Coccinnelle` for each kernel version.

State of practice approach (`apply`) does not provide guarantees on the result.

The `Linux` kernel represents around 20M lines of codes, and the `Coccinelle` rule set applicable to this source code contains 35 semantic patches. Without our proposition, the only way to ensure that no conflicts occured when applying the rules to the source code is to assess that each possible sequence of rule applications (here $35! \approx 10^{40}$) lead to the same result. Considering one second to apply each sequence (an extremely optimistic guess), it leads to approximately 10^{32} years of computation, for a single commit (the `Linux` kernel versioning system contains more than 700K commits in 2018).

One can obviously argue that computation can be parallelised to multiple computers. To tame this issue, we measured in Fig. 5 the average time taken by each semantic patch to analyse and rewrite the source code. One can notice that 75% of the rules are quick to execute (less than 5 minutes in average for each rule, Fig. 5a), and that a few semantic patches (25%, Fig. 5b) are complex and slow. In average, without taking into account the time to orchestrate the rule set, it takes $\approx$ 190 minutes to execute a single sequence on the kernel. It is clearly not reasonable to execute all of the sequences, even in a distributed and parallelized environment. *As a consequence, the current state of practice does not provide any guarantee on the generated source code with respect to rule interactions*.

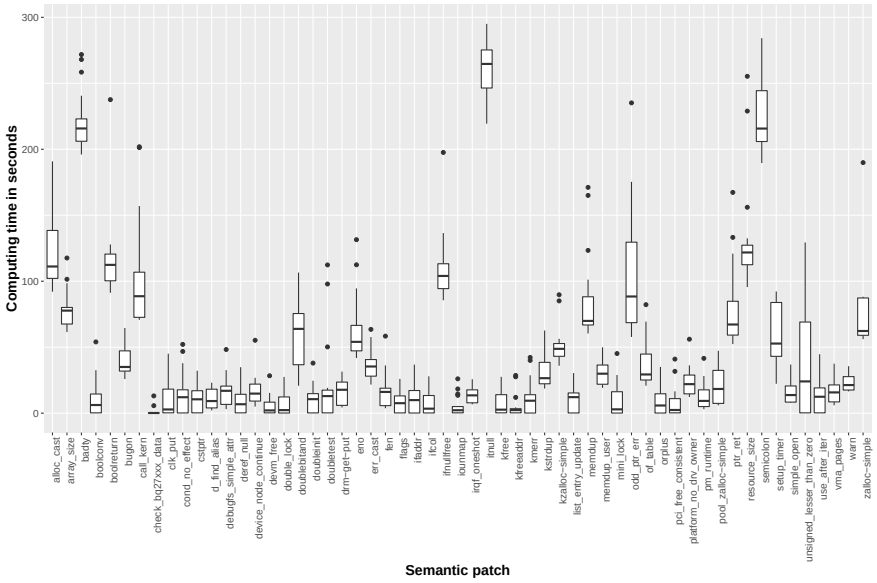
Applying the `iso` operator instead of the `apply` one.

In this section, we show how the `iso` operator described in the previous section is used to identify conflicts among rules in a reasonable amount of time, and how to map the formal model to real-life artefacts. A given rewriter is implemented as the application of the associated semantic patch to the `Linux` kernel. The obtained Δ is implemented by the code diff one can obtain as the output of `Coccinelle` usage. To model the postcondition and detect semantic conflicts, we leverage the following assumption: a semantic patch whose intention is respected, yields an empty diff. Thus, reapplying the rewriting rule to the rewritten kernel must yield an empty diff when the postcondition is respected.

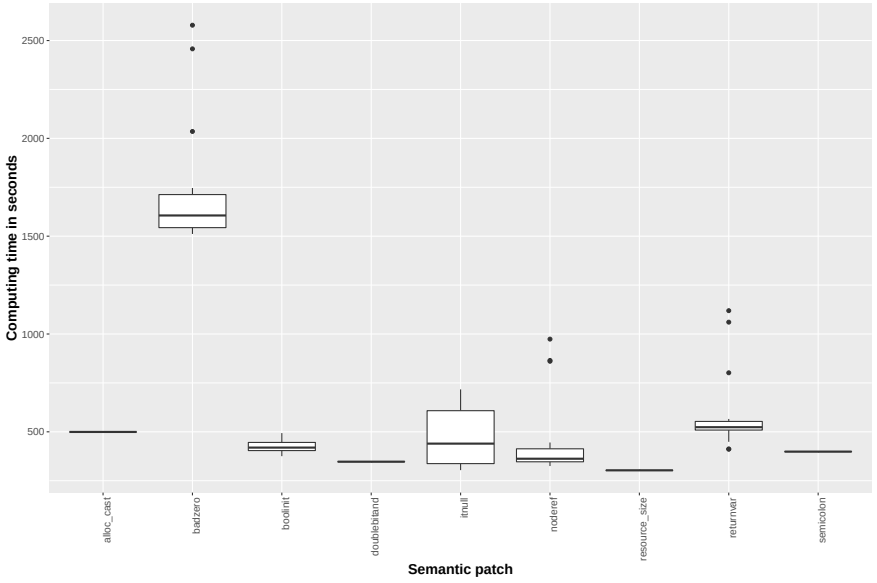
The experimental process applied to each version of the kernel is depicted in Fig. 6. It perfoms the following steps:

1. Checkout the `Linux` source code at a given version (*i.e.*, a given commit),
2. **Phase 1**: apply independently all the 35 semantic patches with `Coccinnelle`, producing code diffs (Δs) to be applied to the kernel to rewrite it;
3. Detect any syntactical conflict among the generated patches. A syntactical conflict is detected when several code diffs target the same lines of `C` code in the kernel;
4. **Phase 2**: For conflict-free situations, apply the code diffs to the source code to obtain the rewritten kernel and verify the 35 postconditions on the obtained kernel.

We depict in Fig. 7 the time consumed by our experimental setup for each of the 19 versions picked in this context. The average execution time is 190 minutes for a given version, with a low standard deviation. Even if this is "long", this time is more reasonable than the absence of guarantee identified in the previous paragraph. We describe in the two following paragraphs how the conflict-detection step identified issues that were previously silenced.



(a) Fast patches (26), applied in less than 5 minutes



(b) Slow patches (9), applied in more than 5 minutes

FIGURE 5 Applying 35 semantic patches to 19 versions of the Linux kernel (execution time)

Validating the absence of syntactical conflicts.

The *conflict?* function, which looks for syntactical issues, is implemented as an intersection between generated deltas of phase 1. As the actions generated are code diffs, they basically consist of a set of *chunks* describing editions (addition or deletion) of the source code. We look then for pairs of chunks (c_1, c_2) such as c_1 will modify the same piece of code than c_2 in the codebase.

No syntactical conflicts were found between the 35 rewriters on the 19 different versions of Linux. That was expected given the high-level of expertise and the limited number of developers involved in the development of such rewriters.

This is a more important result than it might look like at first sight. It means that independently, each semantic patch behaves correctly. Moreover, the 35 patches written by experts do not target the same locations in such a large codebase and do not overlap at the syntactic level. This emphasize the difficulty to identify interactions among rules in such a context.

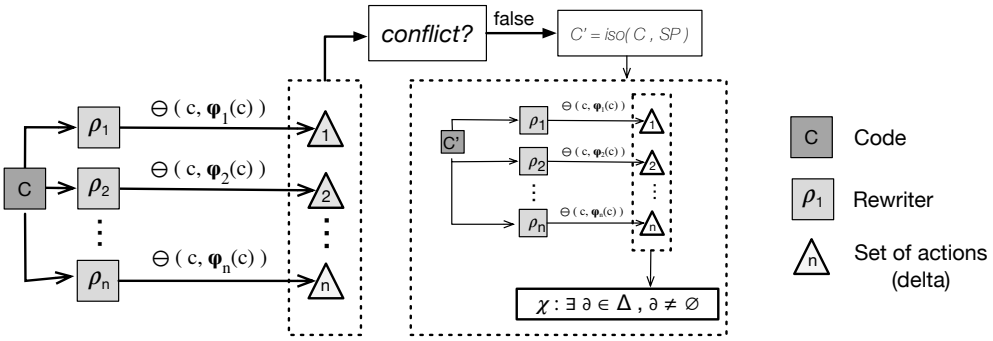


FIGURE 6 Experimental process of the linux use-case

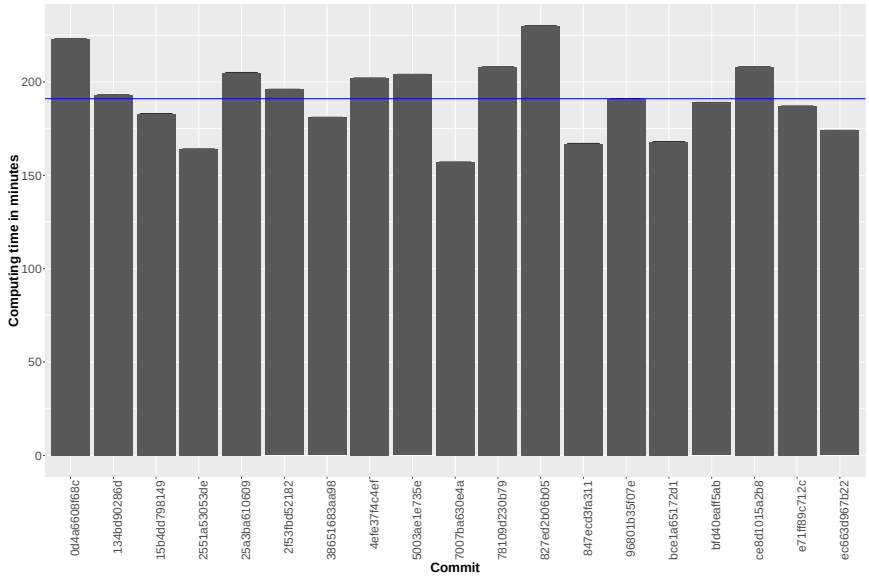


FIGURE 7 Execution time of our proposition in minutes (the line is the average time)

Detecting previously silenced semantic conflicts.

Table 3 lists the interactions detected in the experimental dataset. Out of 19 versions, 7 ($> 36\%$) presented semantic conflicts that were not detected before. The table is ordered in chronological order, meaning that these interactions come-and-go, and are not solved once and for all. From a software engineer point of view, it is interesting to notice how the process helps to debug the rule set. Among 35 fully-functionning semantic patches available, now the developers only have to focus on two of them: `alloc_cast` and `memdup`. They also know the precise location(s) in their code where these two rules conflicts.

According to `Coccinelle` documentation, the `alloc_cast` semantic patch performs the following operation: *"Remove casting the values returned by memory allocation functions like `kmalloc`, `kzalloc`, `kmem_cache_alloc`, `kmem_cache_zalloc` etc."*[6]. The `memdup` patch is self-described in its implementation[7]. It avoids to reimplement the behavior of the `kmemdup` kernel-function at multiple locations in the kernel (which implies `kmalloc` and `kzalloc`). By reading the documentation, one might expect an interaction as both rules target the memory allocation, and it is interesting to notice how fine-grained the issue is. These two rules only conflict when applied to a very specific code subset in the kernel, even if their definition might conflict by essence.

[6]https://bottest.wiki.kernel.org/coccicheck
[7]https://github.com/coccinelle/coccinellery/blob/master/memdup/memdup.cocci
[8]drivers/gpu/drm/amd/powerplay/hwmgr/vega12_processpptables.c
[9]drivers/staging/media/atomisp/pci/atomisp2/css2400/sh_css_firmware.c

TABLE 3 Table of interactions between pairs of semantic patches, on a given line of a specific code file.

commit.id	Rewriter #1	Rewriter #2	File that contains conflict	line of conflict
38651683aa98	alloc_cast	memdup	.../sh_css_firmware.c	146
4efe37f4c4ef	alloc_cast	memdup	.../sh_css_firmware.c	146
b134bd90286d	alloc_cast	memdup	.../vega12_processpptables.c [8]	292
25a3ba610609	alloc_cast	memdup	.../sh_css_firmware.c [9]	133
bce1a65172d1	alloc_cast	memdup	.../vega12_processpptables.c	285
2551a53053de	alloc_cast	memdup	.../vega12_processpptables.c	285
bfd40eaff5ab	alloc_cast	memdup	.../vega12_processpptables.c	292

5.2 | Fixing Energy Anti-patterns in Android applications

In this section, the objective is to focus on a vast and open ecosystem, opposed to the closed and controlled environment provided by the Linux kernel codebase. We analyzed 22 different Android apps publicly available on GitHub. We also took 19 rules that detect and correct when possible energy-related anti-patterns in Android apps [18]. The experiments were run on a mid-2015 MacBook Pro computer, with a 2,5 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 of RAM.

Our goal in this section is not to quantitavely validate each rewriting rule, but to show that issues described in the motivation section of this paper (*e.g.*, overlaping rules) happen in real-life Android applications, on a set of apps that we do not manage. Quantifying how often the rules conflict, on which context and why, is a another contribution, thus out-of-scope of this paper. We selected 22 public Android applications that matched our technical requirements (*e.g.*, Android version compatible with both Paprika, SPOON, and Fenrir tools), thus reducing the number of Android applications analyzed.

We will first focus on the characterization of the overlap that exists among these rules, before diving into a concrete example to see in practice how our contribution properly supports software developers.

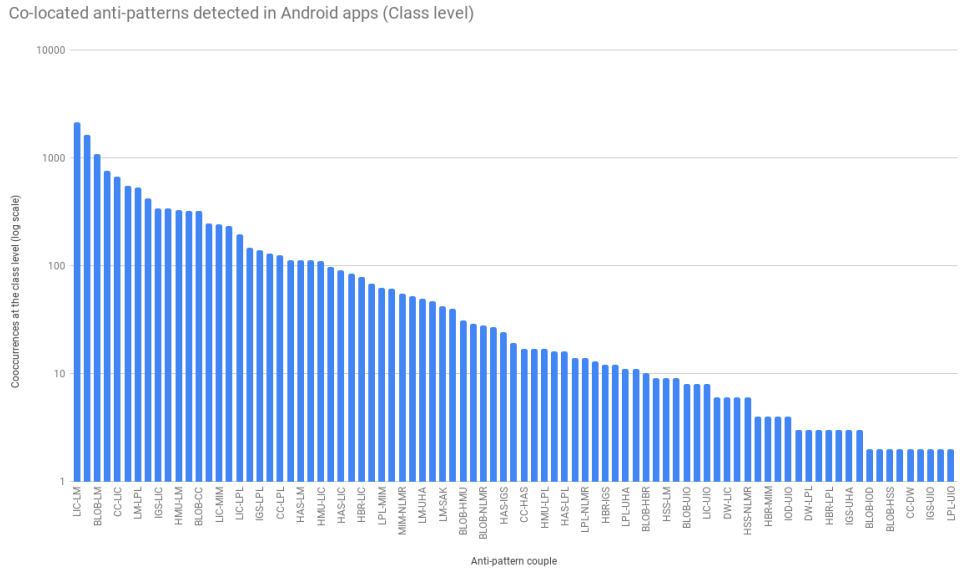
Overlapping Anti-patterns Detection.

Anti-patterns are bad practices in the software design or implementation that can affect quality metrics such as performance, memory, or energy consumption. Mobile applications are by nature critical regarding their energy consumption. Paprika is a tool that allows one to analyse an Android application and detect anti-patterns, including the energy-related ones. Along with these anti-patterns, their respective corrections are developed. According to the toolchain used at the implementation level, the "correction" function is here a function that rewrites the AST of an Android application [14] using Spoon. Thus, fixing multiple anti-patterns at the same time can lead to postcondition violations, especially if they happen at the very same location. We consider here the 22 energy anti-patterns available for detection in the Paprika toolsuite[10]. We use the visualisation tool associated to Paprika logs to identify pairs of co-located anti-patterns. This situation can happen at three different levels: *(i)* the class level, *(ii)* the method level and *(iii)* the instruction level. When several anti-patterns are colocated within the same scope, there is a high probability that repairing the overlapping patterns will interact.

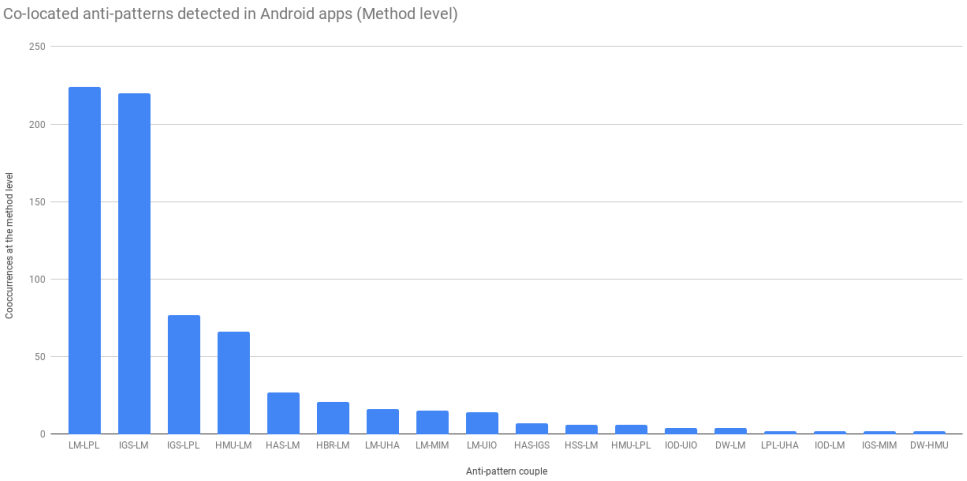
First, to identify the order of magnitude of such interactions, we only consider overlapping pairs. We depict in Fig. 8 the result of analysing the 22 anti-pattern detection rules to the 19 apps of our dataset. At the class level, we detected up to 2,130 co-occurences of the *Leak Inner Class* (LIC [11]) and the *Long Method* (LM [12]) anti-patterns (Fig. 8b). We detected 87 pairs of overlapping anti-patterns at the class level, among the $\binom{22}{2} = 231$ pairs, meaning that almost 40% of the rules overlapped at this level in our dataset. At the method level (Fig. 8b), 18 rules are overlapping, representing 8% of the possible conflicts. At the instruction level, only three anti-patterns interact together. These results strengthen the need to automate the detection of rule interaction issues on concrete examples, as it is not useful to analyse the 231 possible rule combinations but only a small subset of such set.

The previous analysis only considered pairs of anti-patterns. We used the Fenrir tool[13] to visualize at a coarse-grained level the relationship existing among multiple anti-patterns. We represent in Fig. 9 the relationship that exists among anti-patterns. Each node of the graph is an anti-pattern, and the existence of an edge between two nodes means that these two anti-patterns were detected at the very same place in the dataset.

[10]https://github.com/GeoffreyHecht/paprika
[11]Usage of a non-static, anonymous, inner class, leading to memory leak.
[12]Method that has significantly more lines than the other, and can be splitted into smaller ones.
[13]https://github.com/FlorianBourniquel/Fenrir



(a) Overlapping anti-patterns detected at the class level



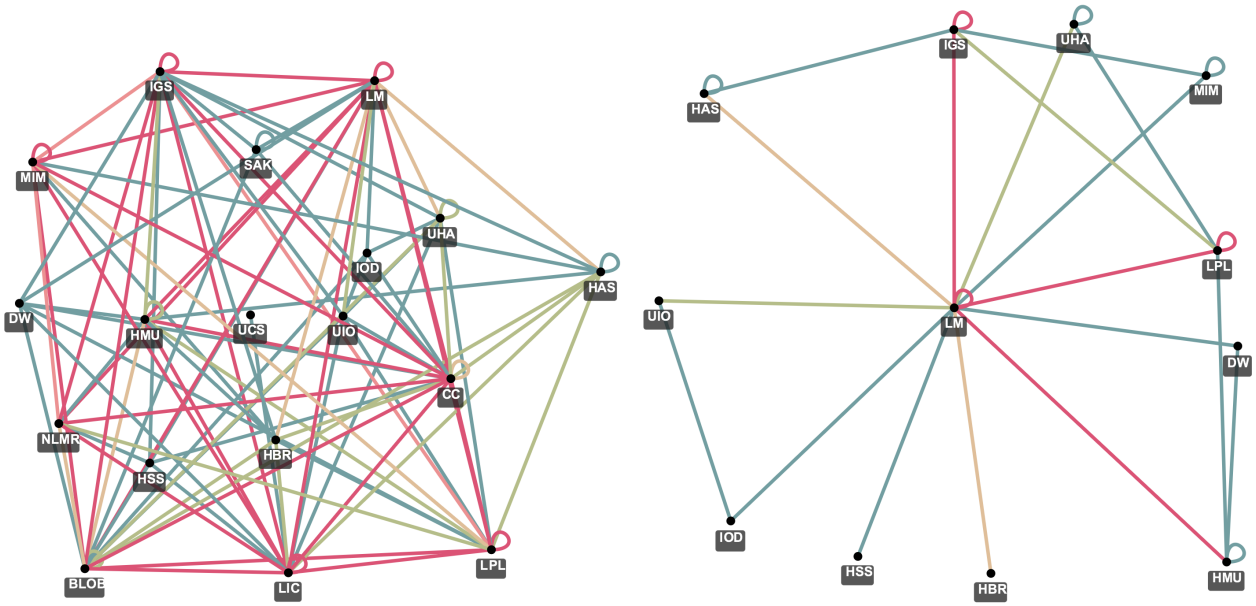
(b) Overlapping anti-patterns detected at the method level

FIGURE 8 Identifying pairs of overlapping anti-patterns in 22 Android apps

Concrete example.

In the previous paragraphs, we validated the existence of overlaps between anti-patterns in existing applications, emphasizing the need for interaction detection mechanisms as the one described in this paper. Unfortunately, it is very difficult to reproduce the build chain associated to these applications (even when the apps rely on tools such as Maven or Gradle), limitating the possibility to fix and rebuild all these apps in an automated way. To tame this challenge and validate the safe reuse of code rewriters in the Android context, we made the choice to perform an in-depth analysis of a single application.

In the Java ecosystem, each rewriting rule is defined as a Spoon `Processor` working at the AST level, and we also used the same mechanism to implement the associated postcondition, as another `Processor` that identifies violations when relevant. To exemplify our contribution on a concrete application, we consider here as a validation example the development of a real Android application. Based on the collaborative catalogue



(a) Class level **(b)** Method level

FIGURE 9 Representing anti-pattern colocations

Android Open Source Apps[14], we selected the *RunnerUp*[15] application. This application is developed by an external team, is open-source, has a large number of installations (between 10,000 and 50,000) and positive reviews in the Android Play Store. From a source code point of view, it has 316 stars on its GitHub repository (December 2017) and has involved 28 contributors since December 2011. It defines 194 classes implemented in 53k lines of code. This application is dedicated to smartphones and smartwatches, thus its energy efficiency is very important.

From the software rewriting point of view, we reused here four different rules. The first one, named R_λ, is used to migrate plain old iterations to the new λ-based API available since `Java 8`, helping the software to stay up to date. The second one, named R_{np}, is used to introduce guards preventing *null* assignments (Fig. 2) in setters, introducing safety in the application. The two others are dedicated to energy consumption anti-pattern fixing: R_h replaces `HashMaps` in the code by a more efficient data structure (`ArrayMaps` are preferred in the Android context), and R_{igs} inlines internal calls to getters and setters (Sec. 2).

We act here as the maintainer of *RunnerUp*, who wants to reuse these four rules. As there is no evident dependencies between the rules, she decides to use the iso operator to automatically improve her current version of *RunnerUp*: $p'_{ru} = \text{iso}(p_{ru}, \{R_{np}, R_{igs}, R_h, R_\lambda\})$.

Figures 10a and 10b show what has been modified by each rule (*i.e.*, the delta they produced). There is no interaction between those two sets of modifications. It happens that all the postconditions hold when applied to p_{ru} and p'_{ru}: *(i)* there is no call to internal getter/setter left in the final program p'_{ru} (Fig.10d), and *(ii)* there is "un-guarded" modification of private field. Thus, the iso operator can be used in this case. The maintainer does not have to wonder about ordering issues *w.r.t.* this set of rules ($4! = 24$ different orders).

To validate the seq operator, we consider a *slightly different implementation* of the R_{igs} rule, named R'_{igs}. This rule rewrites a setter even if it does not contain a single line assignment, and expects as postcondition that the call to the setter is replaced by the contents of the method in the resulting program. With such a rule, $p'_{ru} = \text{iso}(p_{ru}, \{R_{np}, R'_{igs}, R_h, R_\lambda\})$ is not valid with respect to its postcondition, as $\chi'_{igs}(p_{ru}, p'_{ru})$ does not hold, as depicted in Fig. 10d. Actually, the yielded program contains calls to the initial contents of the setter, where the *guarded one is expected* according to this postcondition.

Considering this situation, the maintainer is aware that *(i)* isolated application is not possible when R'_{igs} is involved for p_{ru} and *(ii)* that the conflicting situation might involve this specific rule. She can yield a valid program by calling $\text{iso}(p_{ru}, \{R_{np}, R_h, R_\lambda\})$, meaning that these three rules do not interact together on p_{ru}, and thus an order involving R'_{igs} must be defined. The main advantage of the seq operator is to fail when a postcondition is violated, indicating an erroneous combination that violates the developers' intention. Any call to the seq operator that does put

[14]https://github.com/pcqpcq/open-source-android-apps
[15]https://github.com/jonasoreland/runnerup

```
delete(2)
add(2, "this.column = dbColumn")
```

(a) Delta of the application of `IGS`

```
delete(11)
add(11, "if(name != null) {")
add(12, "this.column = name;")
add(13, "}")
```

(b) Delta of the application of `AddGuard`

```
import org.runnerup.export.format
class DataTypeField {
    private String column = null;
    public DataTypeField(String dbColumn) {
        this.setColumn(dbColumn);
    }
    public void setColumn(String name) {
        this.column = name;


    }
}
```

(c) Initial program

```
import org.runnerup.export.format
class DataTypeField {
    private String column = null;
    public DataTypeField(String dbColumn) {
        this.column = dbColumn;
    }
    public void setColumn(String name) {
      if(name != null) {
        this.column = name;
      }
    }
}
```

(d) Final program

FIGURE 10 Rewriting the RunnerUp Android application (excerpt)

```
import org.runnerup.export.format
class DataTypeField {
    private String column = null;
    public DataTypeField(String dbColumn) {
      if(name != null) {
        this.column = dbColumn;
      }
    }
    public void setColumn(String name) {
      if(name != null) {
        this.column = name;
      }
    }
}
```

FIGURE 11 Final version of the RunnerUp excerpt using the seq operator

R'_{igs} as the last rule will fail, thanks to a postcondition violation. Thus, among 24 different available orderings, the expected one is ensured by calling $p'_{ru} = \text{seq}(p_{ru}, [\ldots, R'_{igs}])$. The expected program is depicted in Fig. 11.

6 | RELATED WORK

Model transformation is *"the automatic manipulation of input models to produce output models, that conform to a specification and has a specific intent"*[19]. Tools such as T-core[20] target the definition and execution of rule-based graph transformations. Such transformation rules are defined as *(i)* a right-hand part that describes the pattern that will trigger the rule and *(ii)* a left-hand part indicating the expected result once the rule has been executed. It does not indicate how to go from the right-hand part to the left-hand, and only expresses the *expected* result. In this paper, we underlined the fact that our rewriting functions are black-boxes that hide their behaviors and inputs (*i.e.*, the right and left parts are hidden). In addition, some rewriting rules implemented in the Android example are not pattern-based and use a two-pass algorithm to catch relevant elements before processing it.

Graph transformation allows one to safely apply graph-transformation rules by assessing if they can be applied in parallel or sequentially and whether the application order matters[21,22,23,24]. It takes the assumption that such transformation rules are well-defined and come with pre- and postconditions formalized. It even allows one to "merge" these rules by providing a new rule that combines the pre- and postconditions of the merge transformation rules. This has some important differences with *our context*: the state-of-practice does not work with well-defined transformation

rules, that are formalized with a pre- nor a postcondition and can come in various format, from various domains, with different languages. Thus, we cannot directly benefit from those outcomes, but try to provide similar properties given our context.

Aspect-Oriented Programming (AOP[25]) aims to separate cross-cutting concerns into *aspects* that will be weaved on a software. Aspects can be weaved in sequence on the same software, thus interactions between different aspects can occur. Their interactions have been identified and studied[26,27]. These works focus on their interaction in order to find a possible schedule in their application to *avoid* any interactions. In this paper, we want to avoid such a scheduling by using the iso composition operator and detect interactions on a given code base. When conflicts are detected, it is possible to reuse aspect-ordering like mechanisms to schedule the application of the rewriting rules.

Transformations can also be directly operated at the code level. Compilers optimize, reorganize, change or delete portions of code according to known heuristics. Works have been done to formalize these transformations and guarantee their correctness[28]. Such toolings, *Alive* for example, are focused on the correctness of a given rule, expressed in an intermediate domain specific language. A strong assumption of our work is that these transformations are black-boxes, correct, bug-free, and we focus on the interactions between rules instead of rule-correctness itself, *making our work complementary to this one*.

Other works focus on concurrent modifications that can occur during a team development. Concurrent refactorings can occur when multiple developers work on the same code and incompatibility can be detected[29]. Such refactoring can be considered as white-boxes graph transformations. Each refactoring is formalized as a function that captures elements in a graph and updates them. This work focuses on refactoring operations only, and needs to formalize and specify the captured inputs of the refactoring (*i.e.*, the pattern that needs to be captured), breaking the black-box assumption of the contribution described in this paper.

Works have been done in *Software Product Lines* (SPL) to support evolution by applying step-wise modifications[30,31]. A modification is brought by so-called *delta modules* that specify changes to be operated on a core module of a SPL. A delta module can operate a finite set of changes in the SPL (*e.g.*, add/remove a superclass, add/remove an interface), and is considered as a white-box function. In addition, conflicting applications of delta modules are solved by explicitly defining an ordering. The sequence of application is explicitly defined by chaining the execution of modules. The white-box paradigm, along the explicit dependency declaration does not match our constraints and initial hypotheses. Finally, the delta-oriented programming of SPL is focused on the safety of a delta module: is a given function safe?, will it bring inconsistencies?, will it perform inconsistent queries? It does not deal with the safe application of multiple delta modules.

7 | CONCLUSIONS & PERSPECTIVES

In this paper, we identified the composition problem that exists when composing multiple rewriting rules using state of practice tools such as `Coccinelle` or `Spoon`. We proposed a formal model to represent rewriting rules in a way compatible with such tools. Through the reification of the deltas introduced by each rule on a given program, we defined two composition operators seq and iso used to safely compose the given rules. The safety is ensured by the validation of postconditions associated to each rule. This enables to detect badly composed rules that would silently ignore developers' intention if sequentially applied. We implemented the model and operators, and applied them on two different real-world applications. The first one is the Linux source-code and its 35 rewriting rules designed to make the kernel evolve along with new features. The second is an external Android application, using four rewriting rules designed to identify and fix anti-patterns, following the latest guidelines from `Google` for Android development.

Contrarily to related work approaches that assume an access to the internal definition of the rewriting rules, we advocate from a reuse point of view the necessity to be fully compatible with state of practice tools that do not expose such information. We intent to extend this work by *(i)* introducing results from the state of the art in the existing tools and *(ii)* applying methods from the test community to the rules. For the former, one can imagine an annotation-based mechanism where a rule would describe in a non-invasive way the elements it selects, as well as the one it produces. Such metadata, when available, will provide a more accurate way to identify conflicts in the general case instead of doing it program by program. It will allow the inference of preconditions based on these data, allowing to bind the proposed approach to classical graph rewriting mechanisms. Using such a binding, it will be possible to empirically benchmark the cost of annotations versus the lack of precision of the blacbox approach proposed in this paper. For the latter, we believe that property-based testing could help to assess rewriting rules composition safety. Keeping the blackbox assumption for rules (*i.e.*, no precondition, and a code that cannot be staticaly analyzed), the only way to assess in controlled condition a set of rewriting rules is to control the program that is used as input of the rewriting process. By generating input programs under given assumptions, it is possible to explore how the rules interact between each other and perform an empirical evaluation of the conflict rate. The work done in the property-based testing community about generators, as well as classical adversarial compilation benchmarks (*e.g.*, LLVM and GCC benchmarks in the C language ecosystem). This might also lead to the reverse engineering of the rules to automatically extract from such applications the selected and rewritten elements.

Acknowledgments

This work is partially funded by the *M4S* project (CNRS INS2I JCJC grant). The authors want to thank Erick Gallesio for his help on kernel development; Geoffrey Hecht for his knowledge of Android optimizations; Mehdi Adel Ait Younes for having developed the initial versions of the Spoon processors; Florian Bourniquel for his tool Fenrir that computes user-friendly graphs of anti-patterns co-occurences; and Mireille Blay-Fornarino and Philippe Collet for their feedback on this paper.

References

1. International Organization for Standardization . *Software Engineering – Software Life Cycle Processes – Maintenance.* https://www.iso.org/obp/ui/#iso:std:iso-iec:14764:ed-2:v1:en; 2006.

2. Lientz Bennett P., Swanson E. Burton. *Software Maintenance Management*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1980.

3. Vlaanderen Kevin, Jansen Slinger, Brinkkemper Sjaak, Jaspers Erik. The agile requirements refinery: Applying SCRUM principles to software product management. *Information and Software Technology.* 2011;53(1):58–70. http://www.sciencedirect.com/science/article/pii/S0950584910001539.

4. Redondo J. M., Ortin F.. A Comprehensive Evaluation of Common Python Implementations. *IEEE Software.* 2015;32(4):76–84.

5. Padioleau Yoann, Hansen René Rydhof, Lawall Julia L., Muller Gilles. Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers. In: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating SystemsACM; 2006; New York, NY, USA. 10.1145/1215995.1216005.

6. Pawlak Renaud, Monperrus Martin, Petitprez Nicolas, Noguera Carlos, Seinturier Lionel. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience.* 2015;46(9):1155–1179. https://hal.inria.fr/hal-01078532.

7. Hecht Geoffrey, Rouvoy Romain, Moha Naouel, Duchien Laurence. Detecting Antipatterns in Android Apps. In: 2nd ACM International Conference on Mobile Software Engineering and Systems:148–149; 2015; Florence, Italy.

8. Klop Jan Willem, others . Term rewriting systems. *Handbook of logic in computer science.* 1992;2:1–116.

9. Mens T.. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering.* 2002;28(5):449–462. 10.1109/TSE.2002.1000449.

10. Berzins Valdis. Software Merge: Semantics of Combining Changes to Programs. *ACM Trans. Program. Lang. Syst.*. 1994;16(6):1875–1903.

11. Falleri Jean-Rémy, Morandat Floréal, Blanc Xavier, Martinez Matias, Monperrus Martin. Fine-grained and Accurate Source Code Differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering:313–324ACM; 2014; New York, NY, USA.

12. Leßenich O., Apel S., Kästner C., Seibt G., Siegmund J.. Renaming and shifted code in structured merging: Looking ahead for precision and performance. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE):543–553; 2017.

13. Rodriguez Luis R., Lawall Julia. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle. In: 11th European Dependable Computing Conference - Dependability in Practice; 2015; Paris, France. https://hal.inria.fr/hal-01213912.

14. Carette Antonin, Younes Mehdi Adel Ait, Hecht Geoffrey, Moha Naouel, Rouvoy Romain. Investigating the energy impact of Android smells. In: IEEE 24th Int. Conf. on Software Analysis, Evolution and Reengineering, Klagenfurt, Austria, February 20-24:115–126; 2017.

15. Blanc Xavier, Mounier Isabelle, Mougenot Alix, Mens Tom. Detecting Model Inconsistency Through Operation-based Model Construction. In: Proceedings of the 30th International Conference on Software Engineering:511–520ACM; 2008; New York, NY, USA.

16. Mosser Sébastien, Blay-Fornarino Mireille, Duchien Laurence. A Commutative Model Composition Operator to Support Software Adaptation. In: 8th European Conf. on Modelling Foundations and Applications:4–19; 2012; Lyngby, Denmark. https://hal.inria.fr/hal-00689706.

17. Benni Benjamin, Mosser Sébastien, Moha Naouel, Riveill Michel. *Conflicts in semantic patch applications in the Linux Kernel.* https://doi.org/10.5281/zenodo.2682531,10.5281/zenodo.2682531; 2019.

18. Hecht Geoffrey, Moha Naouel, Rouvoy Romain. An empirical study of the performance impacts of Android code smells. In: Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016:59–69; 2016.

19. Lúcio Levi, Amrani Moussa, Dingel Juergen, et al. Model transformation intents and their properties. *Software & Systems Modeling.* 2016;15(3):647–684.

20. Syriani Eugene, Vangheluwe Hans, Lashomb Brian. T-Core: A Framework for Custom-built Model Transformation Engines. *Softw. Syst. Model.*. 2015;14(3):1215–1243.

21. Ehrig Hartmut, Pfender Michael, Schneider Hans Jürgen. Graph-grammars: An algebraic approach. In: 14th Annual Symposium on Switching and Automata Theory (swat 1973):167–180IEEE; 1973.

22. Andries Marc, Engels Gregor, Habel Annegret, et al. Graph transformation for specification and programming. *Science of Computer programming.* 1999;34(1):1–54.

23. Heckel Reiko. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science.* 2006;148(1):187–198. http://www.sciencedirect.com/science/article/pii/S157106610600048X.

24. Jouault Frédéric, Allilaire Freddy, Bézivin Jean, Kurtev Ivan, Valduriez Patrick. ATL: A QVT-like Transformation Language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications:719–720ACM; 2006; New York, NY, USA.

25. Kiczales Gregor, Lamping John, Mendhekar Anurag, et al. Aspect-oriented programming. In: Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13:220–242Springer Berlin Heidelberg; 1997; Berlin, Heidelberg.

26. Douence Rémi, Fradet Pascal, Südholt Mario. *Detection and resolution of aspect interactions.* Research Report RR-4435: ; 2002. https://hal.inria.fr/inria-00072153.

27. Tun Thein Than, Yu Yijun, Jackson Michael, Laney Robin, Nuseibeh Bashar. Aspect Interactions: A Requirements Engineering Perspective:271–286. Aspect-Oriented Requirements EngineeringBerlin, Heidelberg: Springer Berlin Heidelberg 2013.

28. Lopes Nuno P., Menendez David, Nagarakatte Santosh, Regehr John. Provably Correct Peephole Optimizations with Alive. *SIGPLAN Not.*. 2015;50(6):22–32.

29. Mens Tom, Taentzer Gabriele, Runge Olga. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science.* 2005;127(3):113–128. http://www.sciencedirect.com/science/article/pii/S157106610500143X.

30. Schaefer Ina, Bettini Lorenzo, Bono Viviana, Damiani Ferruccio, Tanzarella Nico. Delta-Oriented Programming of Software Product Lines:77–91. Software Product Lines: Going Beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. ProceedingsBerlin, Heidelberg: Springer Berlin Heidelberg 2010.

31. Bettini Lorenzo, Damiani Ferruccio, Schaefer Ina. Compositional type checking of delta-oriented software product lines. *Acta Informatica.* 2013;50(2):77–122.

APPENDIX

A CONTENTS OF THE VALIDATION DATASETS

TABLE A1 Composition of our chronological dataset for the Linux validation

Commit ID	Date
ec663d967b22	2017-01-31 16:44:07 +0000
e71ff89c712c	2017-02-02 16:44:14 +1100
5003ae1e735e	2017-03-22 12:01:42 +0100
96801b35f07e	2017-04-20 15:31:26 -0600
38651683aa98	2017-05-24 16:45:36 +0300
4efe37f4c4ef	2017-06-09 11:52:08 +0200
b134bd90286d	2017-07-18 18:51:34 +1000
25a3ba610609	2017-08-22 10:48:53 -0700
0d4a6608f68c	2017-09-20 14:28:52 -0700
78109d230b79	2017-10-21 16:44:50 -0400
2f53fbd52182	2017-11-14 21:17:07 +0900
827ed2b06b05	2017-12-18 11:52:47 -0500
7007ba630e4a	2018-01-29 12:42:15 -0500
ce8d1015a2b8	2018-02-12 09:42:40 +0000
847ecd3fa311	2018-03-19 18:14:26 +0000
15b4dd798149	2018-04-01 00:47:45 +1100
bce1a65172d1	2018-05-17 17:28:09 +0100
2551a53053de	2018-06-04 18:25:05 -0700
bfd40eaff5ab	2018-07-26 19:38:03 -0700

TABLE A2 Composition of our Android dataset, grouped by popularity and ordered by decreasing size (10/09/2018)

Application Name	Description	LoC	Github Stars
signalapp/Signal-Android	Simple private communication with friends	84,303	10,266
nostra13/Android-Universal-Image-Loader	Powerful and flexible library for loading, caching and displaying images on Android	6,006	16,272
cSploit/android	Security app to exploit	29,824	1,882
QMUI/QMUI-Android	Library for speeding up Android development	20,019	5,840
h6ah4i/android-advancedrecyclerview	Fork of ListView	11,395	4,145
evernote/android-job	Android library to handle jobs in the background	10,746	4,668
doggycoder/AndroidOpenGLDemo	OpenGL demo	9,148	1,009
googlemaps/android-maps-utils	Handy extensions to the Google Maps Android API	6,628	2,587
rmtheis/android-ocr	optical character recognition app	4,839	1,851
square/android-times-square	Standalone Android widget for picking a single date from a calendar view	2,324	4,247
ybq/Android-SpinKit-8b088be	Android loading animations	1,971	5,207
alamkanak/Android-Week-View	Calendar view	1,539	2,953
JorgeCastilloPrz/AndroidFillableLoaders	Android fillable progress view working with SVG paths	1,131	1,784
jaredrummler/AndroidProcesses	app for listing running proc	745	1,432
amahi/Android App	Client app for stream	20,833	69
CUTR-at-USF/OpenTripPlanner-for-Android	Open-source trip planner	15,643	113
sagemath/android	Sage client	7,538	67
hyperboria/android	Android app for mesh networking with cjdns	3,318	135
Frank-Zhu/AndroidRecyclerViewDemo Demo	ListView substitute	2,366	708
tekinarslan/AndroidMaterialDesignToolbar	Android Sample Project with Material Design and Toolbar	1,054	709
cyclestreets/android	UK-wide cycle journey planner system	870	161
WuXiaolong/AndroidMVPSample	Android MVP + Retrofit + RxJava2	720	709