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

Benjamin Benni, Sébastien Mosser, Naouel Moha, Michel Riveill

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

Benjamin Benni[1], Sébastien Mosser[1], Naouel Moha[2], and Michel Riveill[1]

[1] Université Côte d'Azur, CNRS, I3S, France
{benni, mosser, riveill}@i3s.unice.fr
[2] Université du Québec à Montréal, Canada
moha.naouel@uqam.ca

Abstract. The tedious process of corrective and perfective maintenance is often automated thanks to rewriting rules using tools such as `Spoon` or `Coccinelle`. These tools consider rules as black-boxes, and compose multiple rules by giving the output of a given rewriting 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. 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)* yield unexpected behaviors that were silently ignored before. We assess this approach by applying rewriting rules used to fix anti-patterns existing in Android applications to external pieces of software available on GitHub.

1 Introduction

It is a commonplace to state that "software evolves", and it is part of software developers' duty to support and operate such an evolution. On the one hand, the adaptive and perfective evolution [8] of a piece of software to address new requirements is taken into account by software development methodologies and project management methods [20]. On the other hand, this evolution [8] is not correlated 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 `Docker` deployment descriptors); code refactoring (*e.g.*, to introduce design patterns) or bugs/anti-patterns correction.

The second form of evolution is usually automated as much as possible, using tools working directly on the source code. For example, migrating from `Python 2.x` to `3.x` is automated using the `2to3` shell command [15]. 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 APIs. To tame this challenge, they develop the `Coccinelle` tool [13], 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 [14] (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-patterns fixing [5]. At runtime, both tools consider rewriting rules as black boxes, applied to a program to generate a patched one.

Contrarily to *abstract rewriting machines* that focus on the confluence, fixed point identification and termination of the rewriting process [7] for a given set of rewriting rules, the previously cited tools took an opposite point of view. They do not consider the confluence of rules application, 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'. Rules are sequentially applied to the input program to yield the final one, passing intermediate results to each others.

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

The main issue with this assumption is the impact of overlapping rules on the yielded 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 rules sequence, and *(ii)* fix the rules or the application sequence to yield the expected result. Classical term rewriting methods cannot be applied to these tools, as it breaks the underlying black-box assumption: 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.

In this paper, **we propose an 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 *delta*s) produced by code rewriters, when the 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 implemented in the `Java` ecosystem, and Sec. 5 describes how the contribution is implemented and then applied to identify rewriting conflicts when automatically patching `Android` mobile applications with respect to `Google` guidelines. 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 two tools that exist in the state of practice (`Coccinelle` and `Spoon`) to automate code rewriting, so to identify the challenges our contribution addresses.

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 [16]. 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`, *l.5*) that is initialized with 0 values (`memset`, *l.8*), and replacing it by an atomic call to `kzalloc` (*l.6*), which do both at the very same time. Wildcards can define guards, for example here the patch cannot be applied if the allocated memory was changed in between (using the `when` keyword). Fig. 1b describes another semantic patch used to fix a very common bug, where the memory initialization is not done properly when using pointers (*l.8*). These two examples are excerpts of the examples available on the tool webpage[3].

Considering these two semantic patches, the intention of applying the first one (R_k) it to use call to `kzalloc` whenever possible in the source code, and the intention associated to the second one (R_m) is to fix bad memory allocation. In the state of practice, applying the two patches, in any orders, does not produce 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, *l.13*), and as a consequence the `kzalloc` optimization is also applied to the fixed `memset`, merging *l.11* and *l.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 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 forgot some of these `kzalloc` calls when it implies badly defined `memset`. Considering p_c, *l.5* and *l.7* are not mergeable until *l.7* 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.

[3] `http://coccinelle.lip6.fr/impact_linux.php`: *(i)* "kzalloc treewide" for R_k and *(ii)* "Fix size given to memset" for R_m.

```
@@
type T;
expression x, E, E1,E2;
@@
- x = kmalloc(E1,E2);
+ x = kzalloc(E1,E2);
... when != \( x[...]=E; \| x=E; \)
- memset((T) x, 0, E1);
```

(a) `kmalloc`∧`memset`$(0) \mapsto$ `kzalloc` (R_k)

```
@@
type T;
T *x;
expression E;
@@

- memset(x, E, sizeof(x))
+ memset(x, E, sizeof(*x))
```

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

```
struct Point {
  double x;
  double y;
};
typedef struct Point Point;

int main()
{
  Point *a;
  // ....
  a = kmalloc(sizeof(*a), 0);
  // not using a
  memset(a, 0, sizeof(a));
  // ...
  return 0;
}
```

(c) Example of a C program (p_c)

```
struct Point {
  double x;
  double y;
};
typedef struct Point Point;

int main()
{
  Point *a;
  // ....
  a = kzalloc(sizeof(*a), 0);
  // not using a

  // ...
  return 0;
}
```

(d) Expected program: $R_k(R_m(p_c))$

Fig. 1: Coccinelle: using semantic patches to rewrite C code

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 let 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`: `CtClass`es contains `CtMethod`s made of `CtExpression`s.

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[4] from null pointer assignment by introducing a test that prevents an assignment to the *null* value to an instance variable in a class. The second one (`IGSInliner.java`, R_{igs}) implements a guideline provided by `Google` when developing mobile application in `Java` using the `Android` framework. Inside a given class, a developer should directly use an instance variable instead of access-

[4] 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`*"*.

```
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 }
```

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

ing it through its own getter or setter (*Internal Getters Setters* anti-pattern). This is one (among others) way to improve the energy efficiency of the developed application with `Android`[5].

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 add 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 processor 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, *l.11*). When introducing the *null* guard, the contents of the `setData` method is changed (Fig. 3c, *l.5-8*), which prevents any upcoming inlining: $R_{igs}(R_{np}(p_j)) = R_{np}(p_j)$. 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.

2.3 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

[5] `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) $p_{np \circ igs} = R_{np}(R_{igs}(p_j))$

Fig. 3: Spoon: applying processors to Java code

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.*, consider their internal decisions as black boxes.

C_1 *Rules isolation.* When rules overlaps, 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 necessary.

C_2 *Conflict detection.* As each rewriter is associated to an *intention*, 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 (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[6]. 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 elsewhere. For a given $p \in AST$, applying φ to it yields an AST p' where $\chi(p, p')$ holds. 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 yielded program.

$$\begin{aligned} &\text{Let } \rho = (\varphi, \chi) \in (\Phi \times X) = P, \quad (\varphi : AST \to AST) \in \Phi \\ &\chi : AST \times AST \to \mathbb{B} \in X, \qquad \forall p \in AST,\ \chi(p, \varphi(p)) \end{aligned} \tag{1}$$

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, leading to a situation where only the last postcondition (χ_1) can be ensured in the resulting program, by construction.

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

Using this operator leads to scheduling issues, as it implies strong assumptions on the functions to be commutative. We build here our contribution on top of two research results: Praxis [2] and a parallel composition operator [12]. 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 $[\alpha_1, \ldots, \alpha_n] \in A^*_<$ yielding m when applied to the empty model. The four kinds of actions available in A are *(i)* the creation of a model element, *(ii)* the deletion of a model element, *(iii)* setting a property to a given value in a model element and *(iv)* setting a reference that binds together two model elements. For example, to build the `Java` program p_j described in Fig. 3a, one can use a sequence of actions S_{p_j} that creates a class, names it `C`,

[6] We do not formalize preconditions, as the tools silently return the given AST when they are not applicable.

creates an instance variable, names it `data`, sets its type to `String`, adds it to `C`, ...

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

Considering a rewriting rule as an action producer introduces the notion of *delta*s 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, for example by relying on patches for `Coccinelle` (*i.e.*, Δs containing elements to add or remove) or on an action model for `Spoon` (*e.g.*, *l.15* in FIG. 2 creates an `if` statement, and *l.20* binds the contents of the setter to this new conditional statement). We model here the application of a sequence of actions to a given AST to modify it, as a generic operator denoted by $\oplus$ (where executing a given action on the AST is a language-specific operation). On the other hand, it is possible for certain languages to define a differentiation tool working at the AST level. For example, the `GumTree` tool [4] exposes the differences between two `Java` ASTs as the minimal sequence of actions necessary to go from the right one to the left one. We denote such a *diff* operation using the $\ominus$ symbol (language-specific).

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

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

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

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 rewriting (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

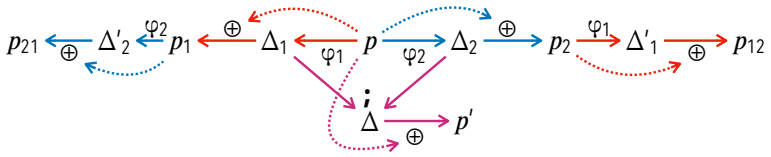

Fig. 4: Sequential ($p \mapsto \{p_{12}, p_{21}\}$) *versus* isolated ($p \mapsto p'$) rewriting

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[7] them into a single one Δ, and *(iii)* apply the result to the initial program. As a consequence, according to this composition semantic, both postconditions χ_1 and χ_2 must hold in the resulting program p' for it to be valid.

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

An interesting property of the isolated composition is to ensure that all postconditions are valid when applied to a program. 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 complements the legacy *apply* one. Using these operators ensures that all the postconditions hold between the initial program and the final one, no matter what happened in between. The *seq* operator implements the sequential composition of an ordered sequence of rules, and the *iso* operator implements the isolated application of a set of rules.

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

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. These two mechanisms address the second challenge of *conflict detection* (C_2, SEC. 2).

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

4.1 Syntactic conflicts as overlapping deltas

Let p 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.

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

On the one hand, the *seq* operator cannot encounter a syntactical conflict, as it it 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[8]: *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 *conflict?* function. 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.

The syntactical conflict detection gives an information to the software developer: among all the rules used to rewrite the program under consideration, there exist a pair of rules that cannot be applied independently. It is still her responsibility to fix this issue, 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 rules, *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

[8] See the Praxis seminal paper [2] for a more comprehensive description of conflict detection in the general case.

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

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, both $apply(p_c, [\rho_k, \rho_m])$ and $apply(p_c, [\rho_m, \rho_k])$ yield a valid result (*i.e.* that do not violate postconditions). However, in the last case, the fixed calls to `memset` introduced by ρ_m make 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 postcondition 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 Implementation & Validation

The approach described in this paper is implemented[9] on top of the `Spoon` framework, in the `Java` ecosystem. Each rule is defined as a `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.

[9] `https://github.com/ttben/ICSR-Implementation-validation`

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	$iso(p_c, \{\rho_{igs}, \rho_{np}\})$	✓	✓	✓

We consider here as a validation example the development of an Android application. Based on the collaborative catalogue *Android Open Source Apps*[10], we selected the *RunnerUp*[11] 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 have 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 piece of 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 `HashMap`s in the code by a more efficient data structure (`ArrayMap`s are preferred in the Android context), and R_{igs} inlines internal calls to getter 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*:

[10] `https://github.com/pcqpcq/open-source-android-apps`
[11] `https://github.com/jonasoreland/runnerup`

$p'_{ru} = iso(p_{ru}, \{R_{np}, R_{igs}, R_h, R_\lambda\})$. It happens that all the postconditions hold when applied to p_{ru} and p'_{ru}, meaning that the *iso* operator can be used in this case. The maintainer do 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, $p'_{ru} = iso(p_{ru}, \{R_{np}, R'_{igs}, R_h, R_\lambda\})$ is not valid with respect to its postcondition, as $\chi'_{igs}(p_{ru}, p'_{ru})$ does not hold. Actually, the yielded program contains call 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 very rule. She can yield a valid program by calling $iso(p_{ru}, \{R_{np}, R_h, R_\lambda\})$, 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 R'_{igs} as the last rule will fail, thanks to a postcondition violation. Thus, among 24 different available ordering, the expected one is ensured by calling $p'_{ru} = seq(p_{ru}, [\ldots, R'_{igs}])$.

Threats to validity. This experiment does not aim to empirically validate the *apply*, *seq* and *iso* operators with a large number of programs and rules. The point here is to validate the expressiveness of the three operators when confronted to a legacy piece of software that was not developed by the authors of the approach. Further experiments are necessary to empirically identify conflicting cases on a large scale code rewriting, measuring the scalabilityof our approach, but is considered out of the scope of this contribution.

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"* [10]. Tool such as T-core [18] targets the definition and execution of rule-based graph transformations. Such transformation rules are defined as *(i)* a right part that describes the pattern that will trigger the rule and *(ii)* a left part indicating the expected result once the rules has been executed. It does not indicate how to go from the right part to the left, and only express 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 uses a two-pass algorithm to catch relevant elements before processing it.

Aspect-Oriented Programming (AOP [6]) 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 has been identified and studied [3,19]. 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 has been done to formalize these transformations and guarantee their correctness [9]. Such tooling, 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 occurs during a team development. Concurrent refactorings can occur when multiple developers work on the same code and incompatibility can be detected [11]. 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 focus on refactoring operations only, and need 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.

Work has been done in Software Product Lines (SPL) to safely evolve it by applying step-wise modifications [17,1]. 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 is 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 hypothesis. 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 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. For the latter, we believe that property-based testing could help to assess rewriting rules composition safety. By generating input programs under given assumptions, it is possible to explore how the rules interact between each others and perform an empirical evaluation of the conflict rate. 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 thanks Erick Gallesio for his help on kernel development; Geoffrey Hecht for his knowledge of Android optmizations; Mehdi Adel Ait Younes for having developed the initial versions of the Spoon processors and Mireille Blay-Fornarino and Philippe Collet for their feedbacks on this paper.

References

1. Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, Mar 2013.
2. Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 511–520, New York, NY, USA, 2008. ACM.
3. Rémi Douence, Pascal Fradet, and Mario Südholt. Detection and resolution of aspect interactions. Research Report RR-4435, INRIA, 2002.
4. Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.
5. Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting antipatterns in android apps. In *2nd ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft 2015, Florence, Italy, May 16-17, 2015*, pages 148–149, 2015.

6. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.
7. Jan Willem Klop et al. Term rewriting systems. *Handbook of logic in computer science*, 2:1–116, 1992.
8. Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.
9. Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. *SIGPLAN Not.*, 50(6):22–32, June 2015.
10. Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani, and Manuel Wimmer. Model transformation intents and their properties. *Software & Systems Modeling*, 15(3):647–684, Jul 2016.
11. Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113 – 128, 2005. Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004).
12. Sébastien Mosser, Mireille Blay-Fornarino, and Laurence Duchien. A Commutative Model Composition Operator to Support Software Adaptation. In *8th European Conference on Modelling Foundations and Applications*, pages 4–19, Lyngby, Denmark, July 2012. SPRINGER LNCS.
13. Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. 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 Systems*, PLOS '06, New York, NY, USA, 2006. ACM.
14. Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.
15. J. M. Redondo and F. Ortin. A comprehensive evaluation of common python implementations. *IEEE Software*, 32(4):76–84, July 2015.
16. Luis R. Rodriguez and Julia Lawall. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle. In *11th European Dependable Computing Conference - Dependability in Practice*, 11th European Dependable Computing Conference - Dependability in Practice, Paris, France, November 2015.
17. Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. *Delta-Oriented Programming of Software Product Lines*, pages 77–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
18. Eugene Syriani, Hans Vangheluwe, and Brian Lashomb. T-core: A framework for custom-built model transformation engines. *Softw. Syst. Model.*, 14(3):1215–1243, July 2015.
19. Thein Than Tun, Yijun Yu, Michael Jackson, Robin Laney, and Bashar Nuseibeh. *Aspect Interactions: A Requirements Engineering Perspective*, pages 271–286. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
20. Kevin Vlaanderen, Slinger Jansen, Sjaak Brinkkemper, and Erik Jaspers. The agile requirements refinery: Applying scrum principles to software product management. *Information and Software Technology*, 53(1):58 – 70, 2011.