Assertions in Intelligent Test Generation May Be Unreliable: A Study on Method Purity Changes in Java Releases

Runze Yu
School of Computer Science
Wuhan University
Wuhan, China

Xiaolu Yao
School of Computer Science
Wuhan University
Wuhan, China

Jifeng Xuan*
School of Computer Science
Wuhan University
Wuhan, China

ABSTRACT

Intelligent test generation techniques, such as EvoSuite, rely on the identification of method purity. Method purity indicates whether executing a method is free of side effects. A method is pure if the state of any object seen by the caller of the method never changes. Identifying method purity can help developers understand program behaviors in many tasks, such as assertion generation for regression testing, path condition inference in static analysis, and state query in synthesis-based program repair. These tasks can be simplified via applying method purity since pure methods cannot modify object states; thus, arbitrarily calling pure methods cannot disturb the current program behaviors. However, method purity can be changed. For instance, in regression testing, purity based assertion generation is unreliable if a pure method is changed into an impure one. In this paper, we conduct an exploratory study on the changes of method purity in Java releases. We refer to such changes as *method purity reversals*. We examine method purity reversals in 282 pairs of Java releases from 10 open-source projects. Our study shows that an originally pure method can be changed into an impure one in the next release and vice versa. Such changes are infrequent but indeed exist. Our study reveals and analyzes the existence of method purity reversals; the method purity reversals should be avoided to reduce the risk to the reliability of intelligent test generation.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Method purity, side effects, test assertions, object-oriented programming, intelligent test generation

ACM Reference Format:
Runze Yu, Xiaolu Yao, and Jifeng Xuan. 2020. Assertions in Intelligent Test Generation May Be Unreliable: A Study on Method Purity Changes in Java Releases . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Corresponding author

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Intelligent test generation techniques, such as an evolutionary testing tool EvoSuite [7], rely on the identification of method purity. In EvoSuite, assertions for regression testing are only generated with pure methods. A method is *pure*, also called *side-effect free*, if the method does not mutate on objects that exist in the pre-states of the method execution [23]. A not pure method is called an *impure* one. Taking a class `List` in Java as an example, a method `size()` of the class `List` that returns the number of elements in the list is pure since it does not mutate the state of the object of `List`; another method `add(Object obj)` is impure since this method adds an element to the object of `List`. Identifying pure methods in a class is an important step to understand program behaviors. In daily development, a developer can easily reuse a pure method without considering the states of objects that have already existed before calling the method; in automated program analysis, a pure method can be safely used in test assertion generation [2, 7] and be directly discarded in model checking [4].

Method purity indicates whether a method is pure or not. Automatic techniques have been proposed to identify the method purity [10, 17, 23, 26]. Fully accurate identification of method purity is infeasible due to the nature of reference inference and external library dependency. Most of the existing techniques have made a trade-off between the scalability and the effectiveness when identifying method purity in practice. Among these existing techniques, ReImInfer by Huang et al. [9, 10] is considered as the state-of-the-art approach, which can be applied to real-world and large-scale programs. Given source files of a Java project, ReImInfer returns a set of pure methods as output via context-sensitive reference inference. The output set of pure methods returned by ReImInfer is a subset of all actually pure methods; that is, ReImInfer may miss several actually pure methods, but all identified methods are accurately pure.

In intelligent test generation, pure methods are used to generate assertions for regression testing. That is, a pure method and its return value (i.e., the state of an object) are converted into an assertion (e.g., `assertEquals(6, list.size())`, where the runtime value of `list.size()` is 6), which is used to detect object changes in its subsequent releases. For instance, in Randoop [19], a tool of feedback-directed random testing for Java, a pure method (called an *observer* method) is used to generate assertions to test changes in the next release; in EvoSuite [8], a tool of evolutionary testing for Java, a pure method taking no parameters and returning primitive values is converted into an *inspector assertion*. Test generation techniques can arbitrarily generate assertions based on pure methods for test cases since calling these methods cannot mutate the current states of objects. These generated assertions can detect whether

a new code change violates the object states during regression testing [8, 19].

However, we observed that there are changes on method purity between two consecutive project releases: an originally pure method may change into an impure method in the next release of a project and vice versa. Assertion generation based on method purity is expected to detect the change of object states in regression testing, but changing from a pure method into an impure one can also mutate the object states. This makes the assertion generation unreliable. We refer to a change on method purity between two releases as a *method purity reversal*.

In this paper, we conducted an exploratory study on method purity reversals, i.e., changes on method purity, in consecutive releases of 10 open-source Java projects. For each project, we extracted releases from the code base and analyzed method purity reversals between two consecutive releases. We explored the existence of method purity reversals and answered three Research Questions (RQs). Our exploratory study shows that there indeed exist method purity reversals in Java releases: method purity reversals appear infrequently and locate in a small number of classes (in RQ1 – existence); in several classes, all method purity reversals are written by the same developers (in RQ2 – contributors); changes from an originally pure method to an impure one prefer adding lines of code while changes from an originally impure method to a pure one prefer deleting lines (in RQ3 – code churns). Our study also presents two cases of observed method purity reversals. The method purity reversal in this study can be viewed as an exceptional scenario of using method purity in test generation and program comprehension.

This paper makes the following major contributions,

- We conducted an exploratory study on method purity reversals in consecutive releases. This study reveals the fact that an originally pure (impure) method in a release may be changed into an impure (pure) one in the next release.
- We combined automatic tools with manual detection to detect method purity reversals and avoided exhaustive manual detection for all methods. We empirically examined the method purity from 282 consecutive release pairs from 10 open-source projects and found 159 method purity reversals.
- We showed that method purity reversals can be found in all 10 projects under evaluation. This adds a threat to the application of intelligent test generation. Meanwhile, we analyzed cases of method purity reversals, the developer contribution, and the code churn to further understand the existence of method purity reversals.

The rest of this paper is organized as follows. Section 2 shows background and motivation. Section 3 presents the study design. Section 4 describes the empirical results of our work. Section 5 presents two cases of observed method purity reversals. Section 6 discusses threats to the validity. Section 7 lists the related work and Section 8 concludes.

2 BACKGROUND AND MOTIVATION

2.1 Background of Method Purity

Identifying method purity can help developers understand program behaviors. Calling a pure method, such as `List.size()` in Java, cannot modify any state of existing program objects while calling an impure method, such as `List.add(Object obj)`, modifies the state of an existing object of `List`. Fig. 1 shows the source code of a pure method `getReverse()` in Release 3.3 of Project `Apache Commons Math` (Project `Math` for short).[1] The method `getReverse()` is pure since it returns a new object of `Line` without modifying any existing object state [12].

```
public Line getReverse() {
    return new Line((angle < FastMath.PI)? (angle + FastMath.PI): (angle
        − FastMath.PI), −cos, −sin, −originOffset, tolerance);
}
```

Figure 1: Source code of a *pure* method in the class `org.apache.commons.math3.geometry.euclidean.twod.Line` from Release 3.3 of Project `Math`.

Arbitrarily adding or removing the call of a pure method never modifies the states of existing objects. In manual debugging, if a developer wants to trace back the root cause that changes the value of an object, the calls of pure methods can be skipped. In intelligent testing and debugging, method purity is used to extract the runtime states of objects. In test generation techniques, such as EvoSuite [7, 8], an assertion is generated for regression testing via collecting the runtime state that is returned by a pure method. EvoSuite [8] uses pure methods that take no parameters and return primitives (e.g., integers or floating-point numbers) to monitor the runtime states of objects. These monitors are called *inspector assertions*. Another test generation technique, GRT [13] uses impure methods to conduct object state fuzzing to produce new states and trigger faults. Besides test generation, method purity can be used in program repair. A synthesis-based repair approach Nopol [25], leverages parameter-free pure methods (called *state query* methods) to extract runtime states of objects, which are used as variables in the synthesis of Boolean expressions.

Existing approaches have worked towards the automatic identification of method purity [10, 16, 17, 23, 26]. Most of the existing approaches can be only applied to small projects or cannot provide accurate identification. Huang et al. [10] have proposed ReImInfer, a static analysis technique based on context-sensitive reference inference. It extends the Java language with three grammatical annotations and can be applied to large-scale projects. The output of ReImInfer is to divide all methods into two categories: pure methods or unknown methods; that is, ReImInfer does not identify impure methods. The method `getReverse()` in Fig. 1 can be directly labeled as a pure method by ReImInfer. ReImInfer can process large-scale Java projects and is considered as the state-of-the-art technique of method purity identification; however, ReImInfer leaves many methods as unknown methods for users since the aim of its implementation is to ensure that all identified pure methods are actually pure without false positive.

[1] Apache Common Math, http://github.com/apache/commons-math/.

2.2 Method Purity Changes

A recent purity analysis tool, Purano, is developed by Yang et al. [26] in 2015. They studied the purity of functions in Java software libraries by using static analysis and found 24–44% of methods in Java libraries are pure. Their tool is implemented for Java bytecode while ReImInfer is implemented to parse Java source code.

The purity of a method may change during software evolution. Ogura et al. [18] developed the first prototype that is used to detect changes on method purity in 2016.[2] Their prototype extracts consecutive revisions for one project and detects method purity with the purity analysis tool, Purano [26]. They evaluated the changes on method purity on 5302 revisions in Project jEdit and 1606 revisions in Project JFreeChart. They found 386 purity changes during the evolution of the two projects.

Differences from the existing work [18]. Our study is different from the existing work on purity changes [18]. Firstly, different from the techniques like ReImInfer in our study, prototypes [18, 26] provide inaccurate results. That is, in their study, an actual pure method can be detected as an impure method. Thus, the study result [18] may not reveal the actual changes in method purity. Secondly, the prototypes [18, 26] are not actively maintained and cannot directly run on the projects in our study. For instance, Johnsson [11] showed that methods with `throw`-statements cannot be fully supported by Purano [26]. Thirdly, our study is conducted in 282 from 10 projects and larger than two projects by Ogura et al. [18].

2.3 Motivation

As mentioned in Section 2.1, method purity is important in manual debugging and automated techniques. For instance, two widely-studied tools of test generation, Randoop [19] and EvoSuite [8], generate assertions for regression testing via collecting runtime states from calling pure methods. In EvoSuite, such assertions are called *inspector assertions*.

An inspector assertion like `assertEquals(state, o.pureMethod())` can be inserted in a test case, where `state` is the runtime state that is returned by calling `pureMethod()` for an object `o`. Such an assertion can be used to detect the state modification in the next release of the same project, i.e., detecting changes in regression testing. In these assertions generated by EvoSuite, pure methods play the role of providing the test oracle, which ensures that the returned object states never change and adding these assertions does not modify objects under test [8, 20].

Method purity is used to generate assertions for regression testing. However, we observed that the method purity is not invariable. Fig. 2 shows the source code of the method `getReverse()` from Release 3.4 of Project `Math`. The source code in Fig. 2 and Fig. 1 belongs to the same method signature, but evolves from Releases 3.3 to 3.4. Different from the pure method in Fig. 1, the method in Fig. 2 is impure because the method assigns a value to the field `reverse`. Calling this impure method in an inspector assertion disturbs existing object states.

During software evolution in two consecutive releases, a method may change from a pure one into an impure one. Since the originally pure method is changed into an impure one, a test generation technique, such as EvoSuite, cannot generate an inspector assertion to detect the change of object states. Arbitrarily calling an inspector assertion may result in extra modification on existing object states; that is, without this inspector assertion, the original states of these objects may be not modified. In this case, the inspector assertions generated by automatic techniques add additional errors to the program during regression testing. This makes the automatically-generated inspector assertions for regression testing unreliable.

[2]In tool demonstration track of ICPC 2016.

```
public Line getReverse() {
  if (reverse == null) {
    reverse = new Line((angle < FastMath.PI)? (angle + FastMath.PI): (
      angle − FastMath.PI), −cos, −sin, −originOffset, tolerance);
    reverse.reverse = this; // Above two lines changed the field
  }
  return reverse;
}
```

Figure 2: Source code of an *impure* method in the class `org.apache.commons.math3.geometry.euclidean.twod.Line` from Release 3.4 of Project `Math`.

The change between Fig. 1 and Fig. 2 motivates us to investigate the existence of changes on method purity. In this paper, we refer to a change on method purity between two releases as a *method purity reversal*. A method purity reversal can be a change from a pure method to an impure one (denoted as *Pure* → *Impure*) or a change from an impure method to a pure one (denoted as *Impure* → *Pure*).

Can a pure method evolves into an impure method in the next release? *Is the change between Fig. 1 and Fig. 2 an accidental observation*? In this paper, we conducted an exploratory study on method purity reversals in 282 release pairs from 10 open-source Java projects. Our study does not focus on the use of method purity in a specific technique like EvoSuite; instead, we aim to explore the existence of method purity reversals.

3 STUDY DESIGN

We present the data preparation, the protocol of detecting method purity reversals, a simple and incomplete algorithm of impure method collection, and three research questions.

3.1 Data Preparation

Our study aims to find out the existence of method purity reversals between releases in Java projects. Table 1 shows the list of 282 release pairs from 10 Java projects in our study. A *release pair* is a pair of two consecutive releases. For instance, Project `math` contains 11 release pairs, which come from 12 consecutive releases. We also present the Lines of Code (LoC) and the number of Java classes in each of the first and the last releases. We choose these 10 projects due to the following reason. We chose three popular projects from the Apache foundation, including Projects `math`, `codec`, and `httpcomponents`. Another two projects `jsoup` and `joda-time` are used in existing studies [6]. The other five projects `plantuml`,

Table 1: List of 10 Projects with Numbers of Release Pairs, LoC, and Classes. Abbreviation of Each Project Name is Marked in Bold.

Project (abbreviation)	#Release pairs ‡	The first release		The last release		List of releases in the study
		LoC	#Classes	LoC	#Classes	
apache/commons-**math** †	11	38370	389	103289	1005	2.0, 2.1, 2.2, 3.0, 3.1, 3.1.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.6.1
apache/commons-**codec**	30	937	14	8109	64	1.1-rc1, 1.1, 1.2-rc1, 1.2, 1.3-rc1, 1.3, 1.4-rc1, 1.4-rc2, 1.4-rc3, 1.4-rc4, ···, 1.11
sqisher/java-**object-diff**	24	2812	56	5830	109	0.8.1, 0.8.2, 0.9, 0.10, 0.10.1, 0.10.2, 0.11, 0.11.1, 0.12, ···, 0.95
apache/**httpcomponents**-client	26	14525	251	39551	575	4.0, 4.0.1, 4.0.2, 4.0.3, 4.1, 4.1.1, 4.1.2, 4.1.3, 4.2, ···, 4.5.2
jhy/**jsoup**	37	2079	25	11057	59	0.1.1, 0.1.2, 0.2.1a, 0.2.1b, 0.2.1, 0.2.2, 0.3.1, 1.1.1, 1.2.1, ···, 1.12.1
jodaorg/**joda-time**	29	25807	154	30666	171	1.5.2, 1.6.0, 1.6.1, 1.6.2, 2.0, 2.1, 2.2, 2.3, 2.4, ···, 2.10.4
plantuml/**plantuml**	37	225044	2653	250959	2781	8059, 2017.08, 2017.09, 2017.11, 2017.12, 2017.13, 2017.14, 2017.15, 2017.17, ···, 2019.11
coobird/**thumbnailator**	29	1461	34	4995	77	0.2.0, 0.2.1, 0.2.2, 0.2.3, 0.2.4, 0.2.5, 0.2.6, 0.2.7, 0.2.8, ···, 0.4.8
cbeust/**jcommander**	34	778	22	2162	46	1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, ···, 1.47
mabe02/**lanterna**	25	5697	79	20983	202	1.0.6, 2.0.0, 2.0.1, 2.0.2, 2.0.3, 2.1.0, 2.1.1, 2.0.4, 2.1.2, ···, 3.0.0
Total	282	317510	3677	477601	5089	

† The source code of all releases in each project can be found from GitHub. For instance, Project `apache/commons-math` can be visited via http://github.com/apache/commons-math/.

‡ A *release pair* is a pair of two consecutive releases. For instance, Project `joda-time` contains 20 consecutive releases, which form 19 release pairs.

`object-diff`, `thumbnailator`, `jcommander`, and `lanterna` are widely-used Java projects. The dataset and experimental results in our study are publicly available.[3]

In purity analysis in Java, several techniques have been developed to identify the method purity, such as JPure [22] and ReImInfer [10]. Several existing tools provide incomplete detection for a set of methods and label the other methods as unknown. For instance, ReImInfer [10] can accurately detect a subset of pure methods and leave the other methods as unknown. In our study, we used ReImInfer as the tool for the detection of method purity.[4] Besides using ReImInfer to identify pure methods, we developed a simple and incomplete algorithm to identify impure methods. Details will be shown in Section 3.2 and Section 3.3.

For each project in the study, we extracted the list of its releases. Our study has not examined all releases due to the following two reasons. On the one hand, releases in the early stage (e.g., implementations that rely on unavailable libraries) may not be locally deployed due to the lack of dependent libraries. For each project in Table 1, we removed any release that is not compatible with the implementation of ReImInfer or cannot be deployed. On the other hand, among 10 projects, Project `joda-time` contains over 60 releases. The large number of releases leads to the large effort of manually checking the method purity. Thus, for Project `joda-time`, we only selected its recent 30 releases that are compatible with ReImInfer (i.e., 29 release pairs) instead of checking all releases. Note that releases in all projects in Table 1 are consecutive releases except one project. In Project `lanterna`, one release is skipped since it cannot be processed by ReImInfer.

3.2 Detecting Method Purity Reversals

To detect method purity reversals between two consecutive releases, we intend to identify the purity of all method signatures that appear in both releases. However, as mentioned in Section 2.1, existing techniques cannot fully identify pure or impure methods in large-scale projects. Due to the data scale, it is infeasible to manually examine the purity of all methods [10, 16, 17, 23, 26]. To balance the manual effort and the number of identified methods, we combine automatic tools with manual examination on the method purity. Note that our study aims to find out the existence of method purity reversals rather than quantify the existence.

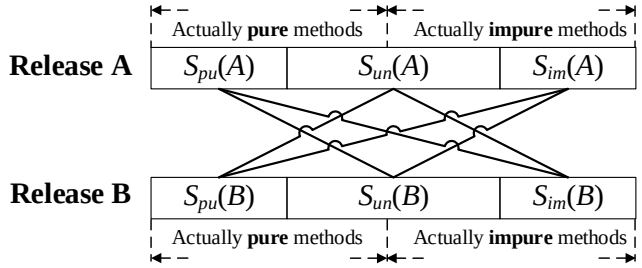
Fig. 3 presents the overview of detecting method purity reversals. In either of two consecutive Releases A and B, the set of all method signatures can be divided into three subsets: a subset of signatures for pure methods that are identified with ReImInfer (i.e., $S_{pu}(A)$ or $S_{pu}(B)$), a subset of signatures for impure methods that are identified with a simple way in Algorithm 1 (i.e., $S_{im}(A)$ or $S_{im}(B)$), and a subset of signatures for other methods (i.e., $S_{un}(A)$ or $S_{un}(B)$). Algorithm 1 will be introduced in Section 3.3. Note that the identification of method purity with ReImInfer or Algorithm 1 is incomplete. For all method signatures in $S_{un}(A)$ or $S_{un}(B)$, the authors of this paper have manually examined the actual purity.

As shown in Fig. 3, s_A and s_B denote the actual purity of a method signature s in Releases A and B, respectively. For instance, given a method signature s appears in both Releases A and B, if s in Release A is identified as a pure method by ReImInfer and s in Release B is unknown, then we manually check the purity of s in Release B, i.e., s_B. If s_B is impure, then we find that there is a method purity reversal of Pure→Impure for s between Releases A and B.

Note that the detection of method purity reversals is incomplete: method purity reversals can be ignored since the intersection of two unknown sets $S_{un}(A)$ and $S_{un}(B)$ is not examined. The workflow in Fig. 3 can be viewed as a trade-off between the huge manual effort of checking method purity reversals and the number of detected method purity reversals on real-world projects. For all method purity reversals that are found in our experiments, the authors of this paper have manually checked and confirmed their changes on method purity.

Table 2 shows the number of methods in each of $S_{pu}(F)$, $S_{un}(F)$, and $S_{im}(F)$ in a final release F of each project. We found that in

[3] Experimental data in the study, http://cstar.whu.edu.cn/p/purity/.
[4] ReImInfer tool, http://github.com/proganalysis/type-inference/.



where s is

$$\begin{cases} \text{Pure} \rightarrow \text{Impure, if } s \in S_{pu}(A) \cap S_{im}(B), \\ \qquad\qquad \text{or } s \in S_{pu}(A) \cap S_{un}(B) \text{ and } s_B = \text{ impure}, \\ \qquad\qquad \text{or } s \in S_{un}(A) \cap S_{im}(B) \text{ and } s_A = \text{ pure}; \\ \text{Impure} \rightarrow \text{Pure, if } s \in S_{im}(A) \cap S_{pu}(B), \\ \qquad\qquad \text{or } s \in S_{im}(A) \cap S_{un}(B) \text{ and } s_B = \text{ pure}, \\ \qquad\qquad \text{or } s \in S_{un}(A) \cap S_{pu}(B) \text{ and } s_A = \text{ impure}. \end{cases}$$

Figure 3: Overview of detecting method purity reversals. Given one release X, $S_{pu}(X)$ is a set of method signatures for pure methods that are identified with ReImInfer; $S_{im}(X)$ is a set of method signatures for impure methods that are identified with Algorithm 1; $S_{un}(X)$ is a set of other method signatures outside $S_{pu}(X)$ and $S_{im}(X)$. For two consecutive releases A and B, s_A and s_B are the actual purity of s in Releases A and B (after manual checking), respectively.

Table 2: Number of Methods in a Final Release F of Each Project

Project	# Methods in the final release			
	$\lvert S_{pu}(F)\rvert$	$\lvert S_{un}(F)\rvert$	$\lvert S_{im}(F)\rvert$	All
math	457	534	208	1199
codec	261	395	51	707
object-diff	299	273	33	605
httpcomponents	732	1954	557	3243
jsoup	457	534	208	1199
joda-time	1486	1996	1168	4650
plantuml	273	14820	1548	16641
thumbnailator	199	141	83	423
jcommander	137	73	59	269
lanterna	810	950	380	2140
Total	5111	21670	4295	31076

seven out of 10 projects, except Projects `codec`, `httpcomponents`, and `plantuml`, the total number of methods in the set $S_{pu}(F)$ of pure methods and the set $S_{im}(F)$ of impure methods is larger than the number of unknown methods in the other set $S_{un}(F)$. This partially shows that the way of detecting method purity reversals (in Fig. 3) is acceptable.

Input: L – a set of all methods in a given release;
Output: R – a subset of impure methods;
1 Initialize two empty sets A and B of methods;
2 $A = A \cup \{m\}$ if in one assignment α in a method $m \in L$, left-value of α is a field or the $this$ reference;
3 $B = B \cup \{m\}$ if in one assignment α in $m \in L$, left-value of α is an object parameter or a field of an object parameter;
4 **while** $|A|$ *increases* **do**
5 Create a new set $C = L \backslash A$;
6 $A = A \cup \{p\}$ if the code of a method $p \in C$ contains $field.m()$ or $this.m()$ where $m \in A$;
7 **end**
8 Return $R = A \cup B$;

Algorithm 1: Incomplete Impure Method Collection

3.3 Simple Algorithm of Impure Method Collection

The design of ReImInfer is to find out an incomplete but accurate set of pure methods. We followed the idea of ReImInfer and proposed a simple algorithm to find an incomplete but accurate set of impure methods. Note that the goal of proposing this algorithm is to reduce the manual effort of examining impure methods rather than focus on the ratio of detected methods.

Algorithm 1 shows a simple and incomplete algorithm of impure method collection. The key idea of this algorithm is to iteratively detect impure methods via checking mutated objects and existing impure methods. The algorithm takes the set of all methods in a given release as input and returns a subset of all impure methods. In Line 2, each method m is added to a subset of impure methods A if the source code of m contains at least one assignment, whose left-value is a field or the `this` reference; that is, the object that invokes m is to be modified. In Line 3, each method m is added to a subset of impure methods B if the source code of m contains at least one assignment, whose left-value is an object parameter or a field of an object parameter; that is, the input parameter is to be modified. In Line 4, another method p is added to the set A if an impure method is invoked by a field or the `this` reference. The algorithm iteratively adds new methods to the set A until no method can be added. All methods collected by Algorithm 1 are impure methods since these methods have modified the existing objects.

Fig. 4 illustrates the usage of Algorithm 1 with a small example. The class `MyObject` consists of five methods. A method `m2()` assigns a value to the field of an object parameter; then `m2()` is labeled as an impure method. A method `m1()` is impure since it assigns a value to the field `str1`. According to Line 6 in Algorithm 1, $m3()$ is identified as an impure method since it invokes an impure method `m1()` with the `this` reference; similarly, `m4()` is also an impure method. For another method `m5()`, it invokes an impure method `m1()` with a newly created object `mo`; then `m5()` is a pure method since it has not modified any existing object states.

3.4 Experimental Setup and Implementation

This study is designed to detect the method purity reversal, i.e., a change between pure and impure on the same method in two

```
public class MyObject {
   String str1;
   String str2;

   void m1 () { str1 = ""; }

   void m2 (MyObject obj) { obj.str1 = ""; }

   void m3 () { this.m1(); }

   void m4 () { this.m3(); }

   void m5 () {MyObject mo = new MyObject(); mo.m1(); }
}
```

Figure 4: Example of applying the incomplete impure method collection. Methods `m1()`, `m2()`, `m3()`, and `m4()` are impure methods that can be identified by Algorithm 1 while the other method `m5()` is pure.

releases. Among all methods in a release, we filtered out two kinds of Java methods. First, we do not consider abstract methods since these methods contain empty bodies and cannot be directly called. Second, we do not consider methods in anonymous classes since an anonymous class does not contain an exact name, which makes the comparison between two releases difficult. We did not filter out inner classes.

In the implementation, we used the Git API to extract the list of releases and commits. We extracted author lists to analyze the contribution of developers. For the identification of method purity, we used an off-the-shelf tool, ReImInfer, to automated collect pure methods [9]. The algorithm in Section 3.3 of impure method collection is implemented on top of Spoon. Spoon is a static analysis library for Java parsing and transformation [21]. We traversed Java classes and extracted the assignments and invocations. The Git diff tool is used to detect the change between two releases for one method signature.

3.5 Research Questions

This study aims to explore the existence of method purity reversals in Java projects. We designed three Research Questions (RQs) to understand method purity under release evolution. These RQs focus on the existence, contributors, and code churns of method purity reversals.

RQ1. *Do method purity reversals between releases exist in Java projects?* We evaluate the existence of method purity reversals in Java projects. In RQ1, we examine the method purity reversals between each pair of two consecutive releases. As mentioned in Section 2.1, several software tasks, such as test case generation, rely on the detection of method purity. There may be potential risks to these tasks if RQ1 shows the evidence that method purity can be changed during software evolution. For instance, assertions in test generation techniques like EvoSuite are unreliable even if one pure method changes into an impure method during regression testing.

RQ2. *How do developers contribute to method purity reversals?* Changes on method purity are written by developers. If method purity reversals add risks to software tasks, RQ2 may help reduce these risks by developers. In RQ2, we aim to find out how many developers are evolved in the commits that change the method purity.

RQ3. *How many lines are changed in method purity reversals?* We investigate the number of changed lines of code, i.e., the code churn [24], in a method purity reversal between two releases. The code churn correlates with software quality [3, 14]. We leverage the code churn to help understand method purity reversals. In RQ3, we measure the size of code changes and observe the distribution of code changes.

4 EMPIRICAL RESULTS

4.1 RQ1. Do method purity reversals between releases exist in Java projects?

We collected method purity reversals from the releases of the projects in the study as well as the numbers of release pairs and classes that contain the method purity reversals. Table 3 shows the count of method purity reversals of both Pure→Impure and Impure→Pure. Among 282 release pairs, we have collected 60 changes on the method purity of Pure→Impure and 99 changes on the method purity of Impure→Pure. This result shows that method purity reversals appear infrequently but indeed exist.

As shown in Table 3, 60 method purity reversals of Pure→Impure exist in 28 classes from 25 release pairs while 99 method purity reversals of Impure→Pure exist in 27 classes from 20 release pairs. This result indicates that most of the observed method purity reversals belong to a small number of classes and release pairs. The observed method purity reversal appears in all 10 projects. This violates the assumption in test assertion generation, such as in Randoop [19] and EvoSuite [8]. These observed method purity reversals can lead to false results for regression testing.

In the category Pure→Impure, the number of method purity reversals in Project `plantuml` reaches the highest value, i.e., 11 from 7 classes; in the category Impure→Pure, the number of method purity reversals in Project `joda-time` reaches the highest value, i.e., 55 from 3 classes. The high value of 55 in Project `joda-time` is caused by the release pair of Releases 2.2 and 2.3: there are 51 method purity reversals in the evolution of this release pair. Meanwhile, in Projects `object-diff`, `thumbnailator`, and `jcommander`, there is no "observed" changes of Impure→Pure.

Note that we do not evaluate the ratio of method purity reversals in all methods. Instead, we focus on the existence: these exist method purity reversals in the 10 projects under evaluation. We will present two observed cases to show the diversity of method purity reversals in Section 5.

There are 159 method purity reversals in 282 release pairs from 10 projects. The method purity reversals appear infrequently and locate in a small number of classes. All projects under evaluation contain method purity reversals: this fact adds a threat to the application of test generation techniques.

Table 3: Number of Release Pairs, Classes, and Methods that contain Method Purity Reversals of Impure→Pure and Pure→Impure

Project	#Release pairs	Pure → Impure			Impure → Pure		
		#Release pairs	#Classes	#Methods	#Release pairs	#Classes	#Methods
math	11	4	5	10	3	8	18
codec	30	2	2	7	1	1	3
object-diff	24	1	1	1	0	0	0
httpcomponents	26	2	2	7	5	5	8
jsoup	37	2	2	8	1	1	2
joda-time	29	1	2	2	2	3	55
plantuml	37	5	7	11	4	4	4
thumbnailator	29	2	2	2	0	0	0
jcommander	34	2	2	5	0	0	0
lanterna	25	4	3	7	4	5	9
Total	282	25	28	60	20	27	99

Table 4: Developers Who Have Contributed to Method Purity Reversals

Project	Pure→Impure		Impure→Pure		All		
	#Methods	#Developers	#Methods	#Developers	#Methods	#Developers in both †	#All developers
math	10	3	18	4	28	2	5
codec	7	2	3	2	10	2	2
object-diff	1	1	0	0	1	0	1
httpcomponents	7	1	8	2	15	1	2
jsoup	8	1	2	2	10	1	2
joda-time	2	1	55	1	57	1	1
plantuml	11	1	4	1	15	1	1
thumbnailator	2	1	0	0	2	0	1
jcommander	5	1	0	0	5	0	2
lanterna	7	1	9	1	16	1	1
Total	60	13	99	13	159	9	18

† *#Developers in both* denotes the number of developers in the intersection of the developer sets for Pure→Impure and Impure→Pure.

4.2 RQ2. How do developers contribute to method purity reversals?

As shown in Section 4.1, most of the method purity reversals locate in not many classes. Therefore, we explore the authors of these method purity reversals, i.e., how many developers have contributed to these changes.

We extracted developers that have changed the method purity according to the author items of Git commits (see Section 3.4). Table 4 lists the number of developers who have contributed to the changes on method purity. We found that all method purity reversals of Pure→Impure are written by 13 developers and all ones of Impure→Pure are written by 13 developers. Meanwhile, there is an overlap between the developers of Pure→Impure and Impure→Pure. There are 8 developers in the overlap; there are 18 developers in total. As shown in Table 4, one developer has written all method purity reversals in several projects. For instance, consider the category of Pure→Impure, there are seven projects (e.g., Project `httpcomponents`) that all method purity reversals are contributed by one developer. This suggests that the changes on method purity may be avoid by training particular developers. Such training can reduce the risk to the reliability of intelligent test generation.

Most of the method purity reversals belong to a small number of classes and are contributed by a small number of developers. In several classes, all method purity reversals are written by the same developers. To avoid the existence of method purity reversals in software evolution, a project team may trace back the commits by these developers or improve the training on the risk of method purity reversals.

4.3 RQ3. How many lines are changed in method purity reversals?

We counted the code churn to measure the size of method purity reversals [1, 3, 14]. In each method purity reversal in one release pair, we counted the number of added lines of code and the number of deleted lines of code by differentiating changes in the method.

Fig. 5 shows the method purity reversals in the categories of Pure→Impure and Impure→Pure. We found that method purity reversals of Pure→Impure locate near the x-axis while method purity reversals of Impure→Pure locate near the y-axis. Most of these method purity reversals are near the left-bottom corner. This result indicates that the number of added lines is more than that of

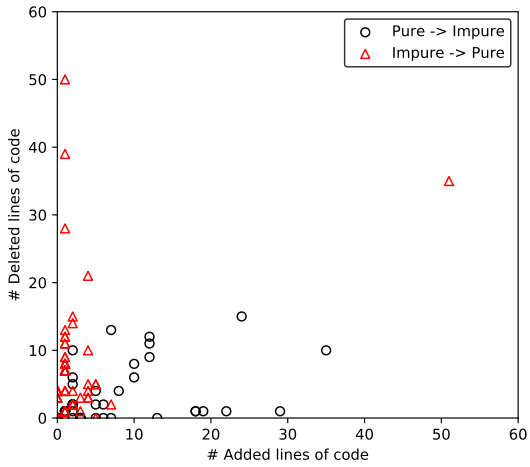


Figure 5: Scatter plot of the numbers of added and deleted lines of code for each method purity reversal.

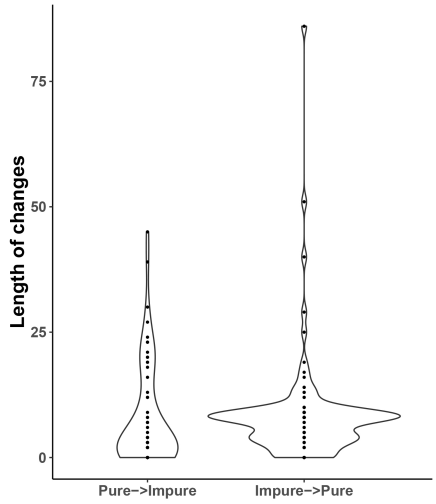


Figure 6: Violin plot of the length of changes for each method purity reversal.

deleted lines if a method changes as Pure→Impure; the number of deleted lines is more than that of added lines if a method changes as Impure→Pure. We also observe that in several changes, the number of lines of code is high. This makes the change complex. As illustrated in Fig. 5, one change adds over 50 lines of code and deletes over 30 lines.

We measured the length of changes via the sum of the number of added lines and the number of deleted lines of code. Fig. 6 presents the distribution of length of changes with a violin plot. In the category of Pure→Impure, the length of zero appears for the most times, i.e., 15 times while in the category of Impure→Pure, eight appears for the most times, i.e., 27 times. Note that there is zero length of changes in Fig. 6 since a change on method purity may be caused by its invoked method whose purity is changed in other

```
public boolean isRepeatable() {
-   for (FormBodyPart part: this.multipart.getBodyParts()) {
-     ContentBody body = part.getBody();
-     if (body.getContentLength() < 0) {
-       return false;
-     }
-   }
-   return true;
+   return getEntity().isRepeatable();
}
```

Figure 7: Method purity reversal of Pure→Impure in the method `isRepeatable()` in the class `org.apache.http.entity.mime.MultipartEntity` from Releases 4.2.1 to 4.2.2 in Project `httpcomponents`.

commits. In the length of change in Fig. 6, we only considered the change in a method with the changes on its invocations.

For method purity reversals of Pure→Impure, the number of added lines is higher than that of deleted lines; for Impure→Pure, the number of deleted lines is lower than that of added lines. In several cases of method purity reversals, the changes of method purity are caused by propagation: the purity of a method is changed since this method invokes another method, whose purity is changed.

5 CASES OF METHOD PURITY REVERSALS

In this section, we present two cases of observed method purity reversals to show the details of these changes.

5.1 Case of Pure→Impure with EvoSuite

We present one case of method purity reversals of Pure→Impure. In this case, EvoSuite generate an assertion for the pure method. However, due to the method purity reversal, this assertion cannot be directly applied in the regression testing. The reason is that the originally pure method is changed into an impure method; meanwhile, the assertion generated by EvoSuite may modify existing states of objects. This makes the regression testing unreliable.

Fig. 7 shows the method purity reversal of Pure→Impure in the method `isRepeatable()` from Releases 4.2.1 to 4.2.2 in Project `httpcomponents`. In the latter release, the added code invokes an impure method `getEntity()`. Thus, the originally pure method is then changed into an impure method.

For Release 4.2.1 in Project `httpcomponents`, EvoSuite generates test cases for Class `org.apache.http.entity.mime.MultipartEntity`. Fig. 8 shows one of these generated test cases. As mentioned in Section 2.1, EvoSuite can generate inspector assertions for pure methods. These assertions are expected to be used in regression testing in the future. EvoSuite only outputs inspector assertions for pure methods since arbitrarily calling a pure method does not modify any object states. As shown in Fig. 8, `assertFalse(mpe0.isRepeatable())` is an inspector assertion for the originally pure method `isRepeatable()`. However, in Release 4.2.2, `isRepeatable()` is changed into an

```
@Test(timeout = 4000)
public void test01()  throws Throwable  {
   [...]
   long long0 = mpe0.getContentLength();
   assertTrue(mpe0.isStreaming());
   assertFalse(mpe0.isRepeatable());   //assertion for isRepeatable()
   assertTrue(mpe0.isChunked());
   assertEquals((−1L), long0);
}
```

Figure 8: A test case generated by EvoSuite for class `org.apache.http.entity.mime.MultipartEntity` in Releases 4.2.1 in Project `httpcomponents`. The object `mpe0` is an object of Class `MultipartEntity`.

impure method. Thus, running the test case for regression testing may directly modify the object state. This makes all assertions after `assertFalse(mpe0.isRepeatable())` unreliable, e.g., `assertTrue(mpe0.isChunked())` in Fig. 8.

5.2 Case of Impure→Pure

Fig. 9 shows Impure→Pure in the method `dataHourMinuteSecond()` from Releases 2.2 to 2.3 in Project `joda-time`. In the former release, the field `dhms` is modified if no initialization is called. In the latter release, the original method body is replaced by a single call that returns a static field value. That is, this method is changed into a pure one.

```
static DateTimeFormatter dateHourMinuteSecond() {
−   if (dhms == null) {
−       dhms = new DateTimeFormatterBuilder()
−           .append(date())
−           .append(literalTElement())
−           .append(hourMinuteSecond())
−           .toFormatter();
−   }
−   return dhms;
+   return Constants.dhms;
}
```

Figure 9: Method purity reversal of Impure→Pure in the method `dateHourMinuteSecond()` in the class `org.joda.time.format.ISODateTimeFormat` from Releases 2.2 to 2.3 in Project `joda-time`.

Summary. In this section, we reported two observed cases of method purity reversals in consecutive releases. These cases can be used to help understand the method purity reversals.

6 THREATS TO VALIDITY

We explained the threats to the validity of our work in three categories.

Threats to construct validity. Our study leveraged both automatic techniques (ReImInfer [10] and Algorithm 1) and manual examination on method purity reversals. As mentioned in Section 3.2, the reason for such combination is that existing automatic techniques cannot fully identify pure or impure methods; however, due to the large number of methods in Table 2, a fully manual examination of all methods is infeasible. Therefore, we designed a combined approach (in Fig. 3) to detect method purity reversals. Identifying method purity in this combined approach is incomplete.

Threats to internal validity. Several method purity reversals in our study are manually detected by two of the authors. We have tried our best to ensure the result of detection. However, there is a threat that the manual detection of method purity reversals contains errors. Our work aims to find out the evidence that there are changes on method purity during release evolution. Besides the result in Section 4, we provide two observed method purity reversals to help readers understand the changes on method purity.

Threats to external validity. We explored the changes on method purity between two consecutive releases from 10 Java projects. There may exist a threat to potential generality. The conclusion in our study may be not applied to other projects. Meanwhile, since the Java language is one of the most popular languages, the existence of changes on method purity in Java releases may not be generalized to other programming languages. Designing a large-scale study across multiple languages could help understand the existence of method purity reversals.

7 RELATED WORK

Sălcianu and Rinard [23] developed a tool named JPPA, short for Java Pointer and Purity Analysis. This tool combines pointer analysis with escape analysis.

Pearce developed JPure [22] in 2011. JPure uses the class inheritance graph and introduces three annotations `@pure`, `@fresh`, and `@local`. JPure cannot parse all calling dependencies and may lead to the low precision.

Huang et al. [9] proposed two techniques for reference inference, ReIm and ReImInfer. ReIm is a type system for reference immutability while ReImInfer is the implementation tool for type inference analysis. The implementation is conducted based on the Checker framework [5]. ReImInfer adds three qualifiers (`@mutable`, `@polyread`, and `@readonly`) to Java programs. This tool first detects the inference on immutable objects and then infers that a method is pure if its parameters, fields, and static class fields are all immutable. ReImInfer is stable and can be applied to large-scale projects.

Different from extending the current program language, Joe-E [15] focuses on a subset of Java. Joe-E has a strict definition of purity, requiring both pure methods that are side-effect free and the only dependencies on inputs. This tool requires that all static fields are final ones and cannot be mutable.

Method purity analysis is also used to narrow down the search space. In Guided Random Testing (GRT) by Ma et al. [13], the test generation method favors the use of impure methods since invoking pure methods cannot modify the object state and is useless to object state fuzzing.

As introduced in Section 2.2, Ogura et al. [18] have developed a prototype for method purity reversals. This prototype is implemented on the top of another tool of method purity detection, Purano by Yang et al. [26]. Both our study and the work by Ogura et al. [18] are incomplete: not all method purity reversals

are detected since automatic techniques are infeasible to fully detect the method purity. The major differences are as follows. First, different from Ogura et al. [18], our result is accurate without false positives. our study is a semi-automatic analysis that combines fully automatic tools and manual analysis. We use ReImInfer [9, 10] to accurately detect pure methods and use our simple algorithm (Section 3.3) to accurately detect impure methods. Second, the study by Ogura et al. [18] is conducted in two projects while our study is conducted on ten projects. Third, we explore the existence, the contributors, and the code churn to help understand the changes on method purity.

8 CONCLUSION

Method purity is crucial to program comprehension in both manual debugging and automated program analysis. In test generation for regression testing, existing techniques such as Randoop and EvoSuite, rely on calling pure methods to detect unexpected behaviors of changes. However, the method purity is not invariable during release evolution. In this study, we conducted an exploratory study on the existence of method purity reversals, i.e., the changes on method purity. We found that method purity reversals exist in all 10 projects in the study. Our study found 159 method purity reversals from 282 pairs of consecutive releases. The ratio of observed method purity reversals is low, but such reversals indeed exist.

REFERENCES

[1] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2017. On the Optimal Order of Reading Source Code Changes for Review. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 329–340. https://doi.org/10.1109/ICSME.2017.28

[2] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 123–133. https://doi.org/10.1145/566172.566191

[3] Jinfu Chen and Weiyi Shang. 2017. An Exploratory Study of Performance Regression Introducing Code Changes. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 341–352. https://doi.org/10.1109/ICSME.2017.13

[4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. 2000. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf (Eds.). ACM, 439–448. https://doi.org/10.1145/337180.337234

[5] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muslu, and Todd W. Schiller. 2011. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 681–690. https://doi.org/10.1145/1985793.1985889

[6] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[7] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 8:1–8:42. https://doi.org/10.1145/2685612

[8] Gordon Fraser and Andreas Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Trans. Software Eng.* 38, 2 (2012), 278–292. https://doi.org/10.1109/TSE.2011.93

[9] Wei Huang and Ana Milanova. 2012. ReImInfer: method purity inference for Java. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 38. https://doi.org/10.1145/2393596.2393640

[10] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. ReIm & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 879–896. https://doi.org/10.1145/2384616.2384680

[11] Mikael Johnsson. 2017. *Purity checking for reference attribute grammars.* Master's thesis. Lund University. http://lup.lub.lu.se/student-papers/search/publication/8972742.

[12] Gary T. Leavens. 2013. Java Modeling Language (JML) Reference Manual, Section 7.1.3. http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_7.html.

[13] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-Analysis-Guided Random Testing. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 212–223. https://doi.org/10.1109/ASE.2015.49

[14] Katsuhisa Maruyama, Shinpei Hayashi, and Takayuki Omori. 2018. ChangeMacroRecorder: Recording fine-grained textual changes of source code. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 537–541. https://doi.org/10.1109/SANER.2018.8330255

[15] Adrian Mettler, David A Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java. In *NDSS*, Vol. 10. 357–374.

[16] Jens Nicolay, Carlos Noguera, Coen De Roover, and Wolfgang De Meuter. 2015. Detecting function purity in JavaScript. In *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*, Michael W. Godfrey, David Lo, and Foutse Khomh (Eds.). IEEE Computer Society, 101–110. https://doi.org/10.1109/SCAM.2015.7335406

[17] Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. 2017. Purity analysis for JavaScript through abstract interpretation. *Journal of Software: Evolution and Process* 29, 12 (2017). https://doi.org/10.1002/smr.1889

[18] Naoto Ogura, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. 2016. Hey! Are you injecting side effect?: A tool for detecting purity changes in java methods. In *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*. IEEE Computer Society, 1–3. https://doi.org/10.1109/ICPC.2016.7503747

[19] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 815–816. https://doi.org/10.1145/1297846.1297902

[20] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), May 20-26, 2007*. 75–84. https://doi.org/10.1109/ICSE.2007.37

[21] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of Java source code. *Softw., Pract. Exper.* 46, 9 (2016), 1155–1179. https://doi.org/10.1002/spe.2346

[22] David J. Pearce. 2011. JPure: A Modular Purity System for Java. In *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 104–123. https://doi.org/10.1007/978-3-642-19861-8_7

[23] Alexandru Salcianu and Martin C. Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3385)*, Radhia Cousot (Ed.). Springer, 199–215. https://doi.org/10.1007/978-3-540-30579-8_14

[24] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Trans. Software Eng.* 37, 6 (2011), 772–787. https://doi.org/10.1109/TSE.2010.81

[25] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[26] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. 2015. Revealing Purity and Side Effects on Functions for Reusing Java Libraries. In *Software Reuse for Dynamic Systems in the Cloud and Beyond - 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 8919)*, Ina Schaefer and Ioannis Stamelos (Eds.). Springer, 314–329. https://doi.org/10.1007/978-3-319-14130-5_22