Genetic Programming and Evolvable Machines (2019) 20:531–580
https://doi.org/10.1007/s10710-019-09355-3

A journey among Java neutral program variants

Nicolas Harrand[1] · Simon Allier[2] · Marcelino Rodriguez-Cancio[3] · Martin Monperrus[1] · Benoit Baudry[1]

Received: 18 December 2018 / Revised: 22 May 2019 / Published online: 25 June 2019

Abstract
Neutral program variants are alternative implementations of a program, yet equivalent with respect to the test suite. Techniques such as approximate computing or genetic improvement share the intuition that potential for enhancements lies in these acceptable behavioral differences (e.g., enhanced performance or reliability). Yet, the automatic synthesis of neutral program variants, through *program transformations* remains a key challenge. This work aims at characterizing *plastic code regions* in Java programs, i.e., the code regions that are modifiable while maintaining functional correctness, according to a test suite. Our empirical study relies on automatic variations of 6 real-world Java programs. First, we transform these programs with three state-of-the-art program transformations: add, replace and delete statements. We get a pool of 23,445 neutral variants, from which we gather the following novel insights: developers naturally write code that supports fine-grain behavioral changes; statement deletion is a surprisingly effective program transformation; high-level design decisions, such as the choice of a data structure, are natural points that can evolve while keeping functionality. Second, we design 3 novel program transformations, targeted at specific plastic regions. New experiments reveal that respectively 60%, 58% and 73% of the synthesized variants (175,688 in total) are neutral and exhibit execution traces that are different from the original.

Keywords Neutral program variant · Program transformation · Java · Code plasticity

1 Introduction

Neutral program variants are at the core of automatic software enhancement. The intuition is that these variants that are different from the original, yet are similar have the potential for enhanced performance, security or resilience. Approximate

✉ Nicolas Harrand
harrand@kth.se

✉ Benoit Baudry
baudry@kth.se

Extended author information available on the last page of the article

computing explores how program variants can provide different trade-offs between accuracy and resource consumption [25]. Software diversity aims at using these variants to reduce the knowledge that an attacker can take for granted when designing exploits [5]. Genetic improvement [28] automatically searches the space of program variants for improved performance.

Despite their key role, the automatic synthesis of neutral program variants, is still a major challenge because of the size of the search space. Starting from one initial program that one aim to improve, there exists a vast amount of possible variants that can be synthesized through small code transformations, most of which do not compile or do not pass the test suite (i.e., ill-formed variants). Exploring this search space randomly can produce a large number of ill-formed variants that are useless for automatic improvement, but still require resources to synthesize and try to compile and test. Our work aims at reducing the number of ill-formed variants that are generated while exploring the space of program variants for automatic improvement tasks. We focus on two specific challenges: understanding *how* and *where* to transform a program to synthesize a neutral variant. The *how* part refers to the design of *program transformations* that introduce some behavioral variations. The *where* refers to the parts of a program that can stand behavioral variations, while maintaining the overall functionality similar to the original program. We call these parts of programs the *plastic code regions*. With the term "plastic" we want to capture a specific characteristic of certain code regions: their "malleability", or they intrinsic capability at being changed to another code while keeping functional correctness, with respect to a given test suite. If we can identify such code regions, they become natural candidates for transformations that aim to synthesize neutral variants. This concept of plastic region is close to the concept of forgiving code regions explored by Rinard [30] or of mutational robustness explored by Schulte et al. [34]. The conceptual difference is that Rinard and Schulte reason about the ability to tolerate perturbations, while, with the term "plastic", we aim at characterizing the ability of the code to exist in multiple forms.

Our work aims to characterize these plastic code regions. This journey focuses on Java programs and the in-depth analysis of various program transformations on 6 large, mature, open source Java projects. We articulate our journey around three main parts. First, we run state of the art program transformations [4, 34] that add, delete or replace an AST node. We consider that a transformation synthesizes a *neutral variant* if the variant compiles and successfully passes the test suite of the original program. This first contribution is a conceptual replication [36] of the work by Schulte et al. [34]. This replication addresses two threats to the validity of Schulte's results: our methodology mitigates internal threats, by using another tool to detect neutral variants, and our experiment mitigates external threats by experimenting with a new set of programs, in a different programming language.

Second, we analyze a set of 23,445 neutral variants. We provide a quantitative analysis of the types of AST nodes and the types of transformations that more likely yield neutral variants. We analyze the interplay between the synthesis of neutral program variants and the specification of the original program provided as a set of test cases. Also, we manually analyze dozens of neutral variants to provide a qualitative analysis of plastic code regions and the role they play in Java programs.

In the third part of our investigation, we design and experiment with three novel, targeted program transformations: ADD METHOD INVOCATION, SWAP SUBTYPE and LOOP FLIP. Our experiments with our 6 Java projects demonstrate a significant increase in the rate of neutral variants among the program variants (respectively 60%, 58% and 73%). We consolidate these results by assessing that the neutral variants indeed implement behavior differences: we trace the execution of these variants, and observe that all neutral variants actually exhibit behavior diversity.

In summary, this work contributes novel insights about neutral program variants, as follows:

- A conceptual replication of the work by Schulte et al. [34] about the existence of neutral variants, with a new tool, new study subjects and a different programming language
- A large scale quantitative analysis of the types of Java language constructs that are prone to neutral variants synthesis with the state of the art program transformations: add, delete and replace AST nodes
- A deep, qualitative analysis of plastic code regions that can be exploited to design efficient program transformations
- Three targeted program transformations that significantly increase the ratio of neutral variants, compared to the state of the art
- Open tools and datasets to support the reproduction of the experiments, available at: https://github.com/castor-software/journey-paper-replication

The rest of this paper is organized as follows. In Sect. 2, we define the terminology for this work. In Sect. 3, we introduce the experimental protocol that we follow in order to investigate the synthesis of neutral variants. In Sect. 4.1, we analyze the types of program transformations and AST nodes that more likely yield neutral variants. In Sect. 4.4, we manually explore and categorize neutral variants according to the role of the code region that has been transformed. In Sect. 4.5, we leverage the analysis of previous sections to design novel program transformations targeted at specific code regions. In Sect. 5, we discuss some key findings of this study. Section 6 elaborates on the threats to the validity of this work, Sect. 7 discusses related work and we conclude in Sect. 8.

2 Background and definitions

Here we define the key concepts that we leverage to explore the different regions of Java programs that are prone to the synthesis of neutral program variants.

2.1 Generic program transformations

Given an initial program, which comes along with a test suite, we consider three generic program transformations on source code that have been defined in previous work [4, 28, 34]. These transformations operate on the abstract syntax tree (AST).

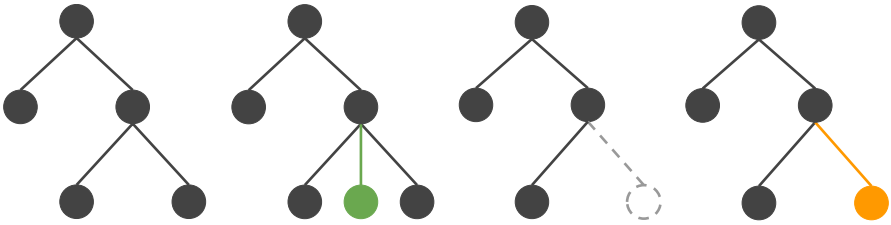

Fig. 1 Generic program transformations

In this context we call *code region* a sub tree present in a program AST. First, we randomly select a statement node in the AST, we check if it is covered by one test case at least (to prevent transforming dead code), then, we consider three types of transformations (cf. Fig. 1).

Definition 1 *Program transformations* We consider the following three transformations on AST nodes

- delete the node and the subsequent subtree (DELETE, Fig. 1c);
- add a node just before the selected one (ADD, Fig. 1b);
- replace the node and the subtree by another one (REPLACE, Fig. 1d).

Definition 2 *Location* The statement at which we perform a program transformation is called the location.

Definition 3 *Transplant* For ADD and REPLACE, the statement that is copied and inserted is called the transplant statement.

This terminology (Definitions 2 and 3) follow a convention established by Barr et al. [2].

We add further constraints to the generic program transformations in order to increase the chance of synthesizing neutral variants. For ADD and REPLACE, we consider transplant statements from the same program as the location (we do not synthesize new code, nor take code from other programs). We also consider the following two additional steps:

- We build the type signature of the location: the list of all variable types that are used in the location and the return type of the statement. The transplant shall be randomly selected only among statements that have a compatible signature.
- When injecting the transplant (as a replacement or an addition to the transplant), the variables of the statement are renamed with names of variables of the same types that are in the scope of the location. Similar restrictions are common in the GI literature, for example Yuan et al. [40] use a type matching based approach (Fig. 2).

Fig. 2 Transplant

```
if (inAvail < max) {
  context.eof = true;
}
```

Fig. 3 Location

```
class A {
  int i = 0;
  void m(int a) {
    boolean b = false;
    [...]
    //Location
    [...]
  }
}
```

Fig. 4 Transformed code

```
class A {
  int i = 0;
  void m(int a) {
    boolean b = false;
    [...]
    if (this.i < a) {
      b = true;
    }
    [...]
  }
}
```

Figure 3 shows an excerpt of program, in which we have selected one location. Figure 2 is a transplant example, i.e., an existing statement extracted from the same program. In order to insert the transplant at the location, we need to rename the variables with names that fit the namespace. The expression `inAvail< max` can be rewritten in 4 different ways: each integer variable can be replaced by one of the two integer variable identifiers (`a` or `i`). The statement `context.eof = true;` can be rewritten in one single way, rewriting `context.eof` into `b` (Fig. 4).

There are different reasons for which a random add or replace fails at producing a compilable variant. Hence we introduce different preconditions to limit the number of meaningless variants.

For REPLACE, we enforce that: a statement cannot be replaced by itself; for both ADD and REPLACE, statements of type *case*, AST nodes of type *variable instantiation*, *return*, *throw* are only replaced by statements of the same type; the type of returned value in a *return* statement must be the same for the original and for its replacement.

2.2 Neutral variant

Given a program P and a test suite TS for P, a program transformation can synthesize a variant program $\tau(P)$, which falls into one of the following categories: (1) $\tau(P)$ does not compile; (2) the variant compiles but does not pass all the tests in TS: $\exists t \in TS | fail(t, \tau(P))$; (3) the variant compiles and passes the same test suite as the

original program: $\forall t \in TS | pass(t, \tau(P))$. This work focuses on the latter category, i.e., all variants that are equivalent to the original modulo the test suite. We call such variants *neutral variants*.

Definition 4 *Neutral variant* Given a program P, a test suite TS for P and a program transformation τ, a variant $\tau(P)$ is a neutral variant of P if the two following conditions hold 1) $\tau(P)$ results from a program transformation on a region of P that is covered by at least one test case of TS; 2) $\forall t \in TS | pass(t, \tau(P))$

This work aims at characterizing the code regions of Java programs where program transformations are the most likely to synthesize neutral variants.

3 Experimental protocol

Program transformations are instrumental for automatic software improvement, and code plasticity is the property of software that supports these transformations. In what follows, we design a protocol to analyze the interplay between transformations, the programming language and code plasticity.

3.1 Protocol

In this paper, we perform the following experiment.

The experiment is budget-based: we try neither to exhaustively visit the search space nor to have a fixed-size sample. Since the investigation of neutral variants is an expensive process, our computation platform is Grid5000, a scientific platform for parallel, large-scale computation [6]. We submit one batch of single program transformations for each program that is run as long as resources (CPU and memory) are available on the grid. Both locations and transplant are selected randomly within the rules detailed in Sect. 2. Then, for each variant that compiles, we extract or compute the metrics described in Sect. 3.3. We also manually analyze dozens of neutral variants in order to build a taxonomy of plastic code regions.

In the second part of our study, we refine the program transformations defined above, in order to target specific code regions. We run another round of experiments to determine the impact of targeted transformations on the neutral variant rate.

3.2 Dataset

We consider the 6 programs presented in Table 1. They were manually selected among popular Java programs (cf[1]) with a strong test suite. All programs are popular

[1] www.mvnrepository.com.

Table 1 Descriptive statistics about our subject programs

	#classes	#stmt	#TC	cov. (%)
commons-lang 3.3.2	132	8442	2514	94
commons-collections 4.0	286	6780	13,677	84
commons-codec 1.10	60	2695	662	96
commons-io 2.4	103	2573	966	87
Gson 2.4	66	2377	966	79
jgit 3.7.0	666	22,333	3341	70

Java libraries developed by either the Apache foundation, Google or Eclipse.[2] The second column gives the number of classes, the third column the number of statements. This latter number approximates the size of the search space for our program transformations. Column 4 provides the number of test case executions when running the test suite and column 5 gives the statement coverage rate. (This number of test case execution corresponds to the number of Junit test methods as reported by maven.)

The programs range between 60 and 666 classes. All of them are tested with very large test suites that include hundreds of test cases executing the program in many different situations. One can notice the extremely high number of test cases executed on commons-collection. This results from an extensive usage of inheritance in the test suite, hence many test cases are executed multiple times (e.g., test cases that test methods declared in abstract classes). The test suites cover most of the program (up to 96% statement coverage for commons-codec). Jgit is the exception (only 70% coverage): it includes many classes meant to connect to different remote git servers, which are not covered by the unit test cases (due to the difficulty of stubbing these servers). This dataset provides a solid basis to investigate the role plastic code regions play to produce modulo-test equivalent program variants.

3.3 Metrics

Definition 5 *Neutral Variant Rate (NVR)* is the ratio between the number of neutral variants and the number of transformations that produce a variant that compiles: *#NeutralVariants/#Compile.*

The neutral variant rate is a key metric to capture the plasticity of a code region: the higher it is for a certain region, the more this region can be used by program transformations to synthesize valid variants. It is designed to consider only variants that compile, because (1) our goal is to study what characteristics impact a program tolerance for alternative implementation (2) we compare it for transformations with widely different *#CompileVariants/#Transformation* ratios. This ratio is more linked

[2] The exact versions of the library and the whole dataset is available here: https://github.com/castor-software/journey-paper-replication/tree/master/projects.

to the transformation implementation than to whatever characteristic of the targeted program region. It is noteworthy that running tests is the actual costly part of the search for neutral variants. Non-compilable variants fail fast and therefore, do not cost much search time.

We collect the following metrics to characterize the regions where we perform program transformations.

Definition 6 *Location features* Let us call *loc* the location yielding the neutral variant. We focus on the following features: (1) TC_{loc} is the number of test cases that execute *loc*. (2) $Transfo_{loc}$ is a categorical feature that characterizes the type of transformation that we performed on *loc*: ADD, DELETE or REPLACE. This can be further refined by considering the type of AST node where the transformation occurs.

3.4 Research questions

Our journey among neutral variants is organized around the following research questions:

RQ1. To what extent can we generate neutral variants through random program transformations?

This first question can be seen as a conceptual replication of Schulte et al.' [34]'s experiment demonstrating software mutational robustness. Here, we analyze the same phenomenon with a new transformation tool, new study subjects and in a different programming language.

RQ2. To what extent does the number of test cases covering a certain region impact its ability to support program transformations?

This question addresses the interplay between the synthesis of neutral variants and the specification for specific code regions. Since our notion of neutral variant is modulo-test, we check if the number of test cases that cover the location influences the ability to synthesize a neutral variant.

RQ3. Are all program regions equally prone to produce neutral variants under program transformations?

In this question, we are interested in analyzing whether the type of AST node or the type of transformation has an impact on the neutral variant rate. For instance, it may happen that loops are more plastic than assignments. We study three dimensions in the qualification of transformations: (1) how they are applied (addition of new code versus deletion of existing code); (2) where they are applied, i.e. the type of the locations (e.g. conditions versus method invocations); and (3) for ADD and REPLACE, the type of the transplant.

RQ4. What roles the code regions prone to neutral variant synthesis play in the program?

This question relies on a manual inquiry of dozens of neutral variants from all programs of our dataset, to build a taxonomy of program neutral variants. Here, we

categorize different roles that certain code regions can play (e.g., optimization or data checking code) and relate this role to the plasticity of the region.

RQ5. Can program transformations target specific plastic code regions in order to increase their capacity at synthesizing neutral variants that exhibit behavioral variations?

We exploit the insights gained in RQ3 and RQ4 to define novel types of program transformations, which refine the ADD and REPLACE generic transformations: ADD METHOD INVOCATION, SWAP SUBTYPE, LOOP FLIP. These transformations perform additional code analysis to select the location. This question investigates whether this refinement helps to reduce the number of variants that are not neutral program variants hence cannot be used as candidates for modulo test equivalent improvement.

3.5 Tools

To conduct the experiments described in this paper, we have implemented a tool that runs program transformations on Java programs and automatically runs a test suits on the variant, in order to select neutral variants. This tool, **Sosiefier** is open source and available online.[3] The analysis and transformation of the JAVA AST mostly relies on another open source library called Spoon [27].

To capture, align and compare execution traces described in Sect. 4.5, we have implemented **yajta**,[4] a library to tailor runtime probes and trace representations. It uses a Java agent, which instruments Java bytecode with Javassist [9], to collect log information about the execution. Scalability is a key challenge here, since the insertion of probes on every branch of every method represents a considerable overhead both in terms of execution time, and heap size. For example, a single test run can generate a trace up to GBs of data, which turns into a performance bottleneck when comparing the traces from hundreds of variants. This is especially true for performance test cases such as `PhoneticEnginePerformanceTest` (335,500,702 method calls and 990,617,578 branches executed) in `commons-codec`. These issues are well described in the work of Kim et al. [20].

Consequently, we optimized the tracing process as follows: (1) execute and compare only the test cases that actually cover the location in the original program ; (2) add transformation-specific knowledge to target the logs (e.g. the addition of a method invocation only requires to trace method call) ; and (3) collect and store complete traces only for the original program, and compare this trace with the variant behavior *on-the-fly*. This way, we determine, at runtime, if a divergence occurs and we do not need to store the execution trace of the variant.

[3] https://github.com/DIVERSIFY-project/sosiefier.

[4] https://github.com/castor-software/yajta.

Table 2 Neutral variant rate for the synthesis of neutral program variants with the generic, random program transformations

	add	del	rep	NVR (%)	Exploration (%)
commons-codec	289	146	266	18.0	91.9
commons-collections	3912	754	3960	21.8	83.3
commons-io	1754	319	1472	21.1	92
commons-lang	419	190	537	15.7	78
gson	2199	215	1897	25.3	80.3
jgit	1924	1375	2963	30.0	57
total	10,078	2809	10,558	23.9	–

4 Results

4.1 Neutral variant rate of random transformations

This section focuses on RQ1.

RQ 1 To what extent can we generate neutral variants through random transformations?

We run program transformations on our six case studies (cf. Table 1). Table 2 gives the key data about the neutral variants computed with the budget-based approach described in Sect. 3.1. It sums up the results of the 180,207 variants generated, from which 98,225 compile and 23,445 are neutral variants. The second, third and fourth columns indicate the number of neutral variants synthesized by ADD, DELETE or REPLACE. The fifth column indicates the global neutral variant rate (NVR) as defined in Definition 5, i.e., the rate of neutral variants among all variants that we generated and that compile. The last column (exploration) indicates the rate of program statements on which we ran a transformation, i.e., the extent to which we explored the space of locations. The low exploration rate for `jgit` is related to the large size of the project: since our exploration of program transformations has a bounded resource budget, we could not cover a large program as much as a small one.

The data in Table 2 provides clear evidence that it is possible to synthesize neutral variants with program transformations. In other words, it is possible to transform statements of programs and obtain programs that compile and are equivalent to the original, modulo the test suite.

The program variants that compile are neutral variants in up to 30% of the cases (for `jgit`).

This first research question is a conceptual replication of the study of Schulte et al. [34]. Their program transformations are the same as ours. Yet, they ran experiments on a very different set of study subjects: 22 programs written in C, of size ranging from 34 to 59K lines of code and with test suites of various coverage ratios

(from 100% to coverage below 1%). They also run experiments on the assembly counterpart of these programs. Their results show that 33.9% of the variants of on C code are neutral, with a standard deviation of 10. They also obtain 39.6% of neutral variants at the assembly level, with a standard deviation of 22 on assembly variants.

Our results confirm the main observation Schulte and colleagues: running ADD, DELETE and REPLACE randomly can synthesize a significant ratio of neutral program variants. The neutral variant rate between both our and Schulte's experiments are of the same order of magnitude. Their experiments generate slightly more neutral variants, which could indicate that different programming languages allow various degrees of plasticity. In particular, a stronger type system can limit code plasticity. Yet, the in-depth analysis of differences between languages is outside the scope of this paper.

Answer to RQ1: Program transformations, applied in random code regions, can synthesize neutral program variants on Java source code. The ratio of neutral variants varies between 15.7% and 30.0%, out of thousands of variants, for our dataset. These new results confirm the main observations of Schulte and colleagues.

4.2 Sensitivity to the test suite

RQ 2 To what extent does the number of test cases covering a certain region impact its ability to support program transformations?

Here, we check if the number of test cases that cover a statement affects the plasticity that we observe. In other words, we evaluate the importance of the number of test cases that cover a location with respect to the probability of synthesizing a neutral variant when we transform that point with one of our program transformations.

In order to analyze this impact, we look at the distribution of neutral variant rate for all trials made on statements covered by a given number of test cases. Yet, in all projects, the distribution of statements according to the number of test cases that cover it is extremely skewed: more than half of the statements are covered by only one test case and then there is a long tail of few statements that are covered by tens and even hundreds of test cases.

Figure 5 represents the following information, given any location at which we synthesized one or multiple variants that compile, what is the probability that we succeed in getting a neutral variant, given the number of test cases that cover the location? Because of the skewed distribution of statements with respect to the number of covering test cases, we group data in bins of locations that represent at least 4000 transformations. Bins for low numbers of test cases cover a narrower range of values because statements covered by few tests are more common than statements covered by a large amounts of tests.

The broken line represents the average neutral variant rate per bin of locations. Boxes represent the first and last quartile and the median for the distribution of neutral variant rate for statements covered by n test cases. Circles represents outliers (outside of a 95% confidence interval) statement for each classes. For

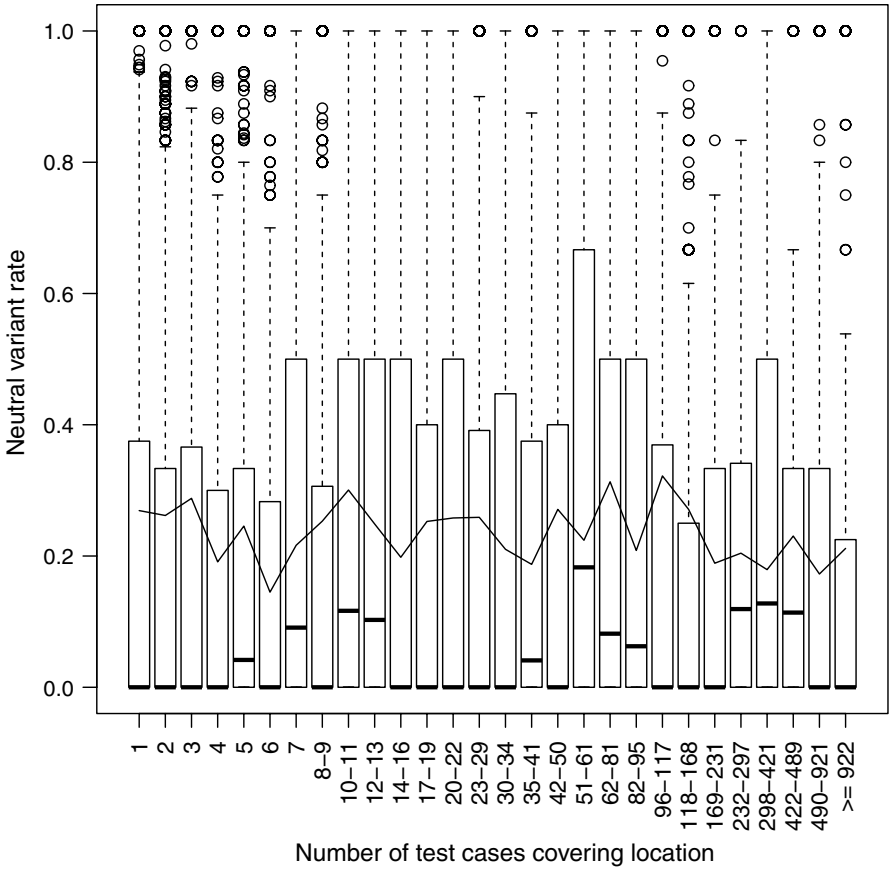

Fig. 5 Neutral variant rate with respect to the number of test cases covering a location

example, for the 5943 locations covered by 1 test case, the weighted average neutral variant rate is 26.9% and 25% of these points support the synthesis of neutral variants in more than 37.5% of the trials. Outliers are locations for which neutral variant rate is above 93.8%.

For 17 out of 28 bins, the median neutral variant rate is 0%, meaning that, for at least half of the locations, none of the variants tried are actually neutral. Meanwhile, the first quartile is above 0% for all bins. This means that we successfully synthesized neutral variants for at least 25% of statements covered, independently of the amount of test cases (for 11 bins it is actually more than 50% of statements). The average neutral variant rate is close to the overall neutral variant rate of 23.9%, whatever the number of test cases covering the location.

Under the assumption of a linear model, the part of the neutral variant rate explained by the number of test cases is negligible (Adjusted R-squared: 0.002036). This implies either that the ability to synthesize a neutral variant on

Table 3 Distribution of statement type across projects

Node Type	Min	Med (%)	Max (%)
Invocation	34%	37	39
Assignment	17%	19	22
Return	10%	13	19
If	9.4%	10	14
ConstructorCall	4.2%	6.6	8.8
UnaryOperator	3.1%	3.8	8.6
Throw	1.7%	2.9	4.4
Case	0.13%	1.2	2.6
For	0.55%	0.76	1.5
ForEach	0.37%	0.72	0.87
Try	0.17%	0.65	1.4
While	0.40%	0.62	0.85
Break	0.18%	0.54	1.6
Continue	0.018%	0.21	0.65
Switch	0.033%	0.17	0.32
Synchronized	0	0.048	0.21
Enum	0	0.042	0.094
Do	0	0.032	0.091
Assert	0	2.68e−03	0.0014

a given statement is not significantly influenced by the number of test cases that cover it with a linear model.

Answer to RQ2: the number of test cases that cover a location is independent from the ability to synthesize a neutral variant at this point. We believe that this indicates the presence of inherent *code plasticity*, a concept for which we propose a first characterization in the RQ5. To some extent, the neutral variant rate on locations that are covered by large numbers of test cases reflects this amount of software plasticity.

4.3 Language level plasticity

RQ 3 Are all program regions equally prone to produce neutral variants under program transformations?

As a preliminary step for our analysis of the plasticity of language structures, we analyze the usage frequency of each construct. Table 3, summarizes the usage distribution of each construct listed by decreasing median frequency. It appears that 6 constructs are frequently used, in approximately the same proportion in all projects (the top 6 lines of the table). There is no surprise here: these constructs correspond to the fundamental statements of any object-oriented program (`assignment`, `if`, `invocation`, `return`, `constructor call` and `unaryOperator`).

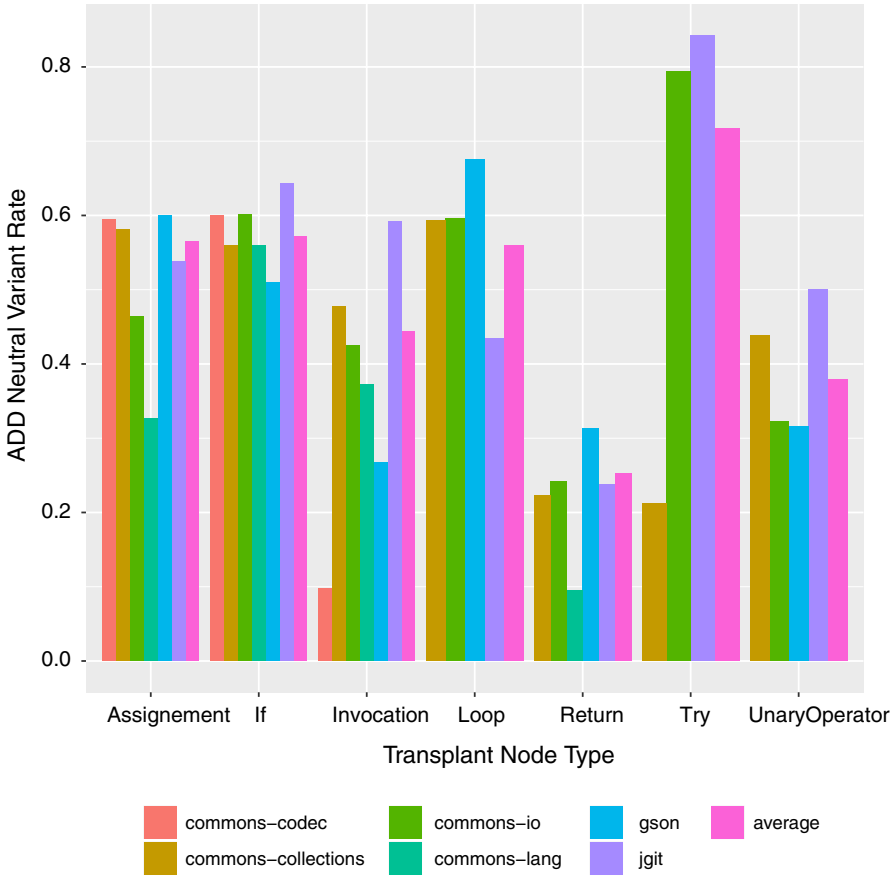

Fig. 6 Neutral variant rate for the ADD transformation, depending on the type of AST node used as transplant

The 13 other constructs present in the table are an order of magnitude less frequent than the top constructs. They are also used in more various ways across programs. For instance, commons-collections favors `for-each` and `while` loops, while commons-codec uses `for` loops. This can be explained by the different types of structure that these projects use: collections vs arrays.

The use of `switch` and its child nodes (`break`, `case`, and `continue`) as well as `try` are also unequally distributed across projects. This disparity partly explains the variation in the observations presented in the following section: uncommon constructs lead to more variations.

4.3.1 ADD

Figure 6 displays the neutral variant rate of the ADD transformation according to the type of statements added (type of the transplant node in the AST). Each cluster of

bars includes one bar per case study. The darkest bar represents the average neutral variant rate. The figure only displays the distributions for the node types for which we performed more than 25 trial transformations for a given project.

The first striking observation is that neutral variant rates reach significantly high values. In four cases, the random addition of statements yields more than 60% neutral variants: add "if" nodes in `jgit`, add "loop" nodes in `gson` and "try" nodes in both `commons-io` and `jgit`. The addition of such nodes provides important opportunities to explore alternative executions.

We observe important variations between node types as well as between projects. However, some regularities emerge: for instance, adding a "return" always yields a low neutral variant rate. This low plasticity of return statements matches the intuition: this is the end point of a computation and it is usually a region where a very specific behavior is expected (and formalized as an assertion in the test). Meanwhile, the addition of "Try" statements appears as an effective strategy to generate neutral variants.

Looking more closely at Fig. 6, we realize that on average, the addition of "assignment" nodes is the most effective (if we exclude addition of "try" nodes for which we don't have enough data for all projects). This can be explained by the fact that there are many places in the code where the variable declaration and the first value assignment for this variable are separated by a few statements. In these situations it is possible to assign any arbitrary value to the variable, which will be canceled by the subsequent assignment. Yao et al. [39] observed a similar phenomenon of specific assignments that "squeezes out" a corrupted state. Also, for some projects, such as `commons-io` and `jgit`, the addition of "invocation" nodes is effective. It probably indicates a non-negligible proportion of side-effect free methods in the program, but further experimentation on that matter is detailed in Sect. 4.5.1.

The addition of conditionals and loops is also effective. It is important to understand that a large number of these additional blocks have conditions such that the execution never enters the body of the block, meaning that only the evaluation of the condition is executed.

4.3.2 DELETE

Figure 7 shows the neutral variant rate of the DELETE transformation in function of the type of the AST node deleted, grouped by project. The figure only shows the node types for which enough data were collected (More than 25 transformations tried for a given project). While we observe large variations between projects for a given node type, we also note that there is a large variation in the neutral variant rate per node type. For instance, this figure suggests that method invocations are less specified than while-blocks, since the neutral variant rate is higher.

It appears that deleting a method invocation produces above average results for all projects of our sample. We explain this effect by the presence of side-effect free methods which can be safely removed (discussed also in the next section) and by the existence of many redundant calls (discussed in the next section).

The deletion of "continue" nodes is quite effective at synthesizing neutral variant as it yields 27% success overall (Not included on the graph since not enough trials

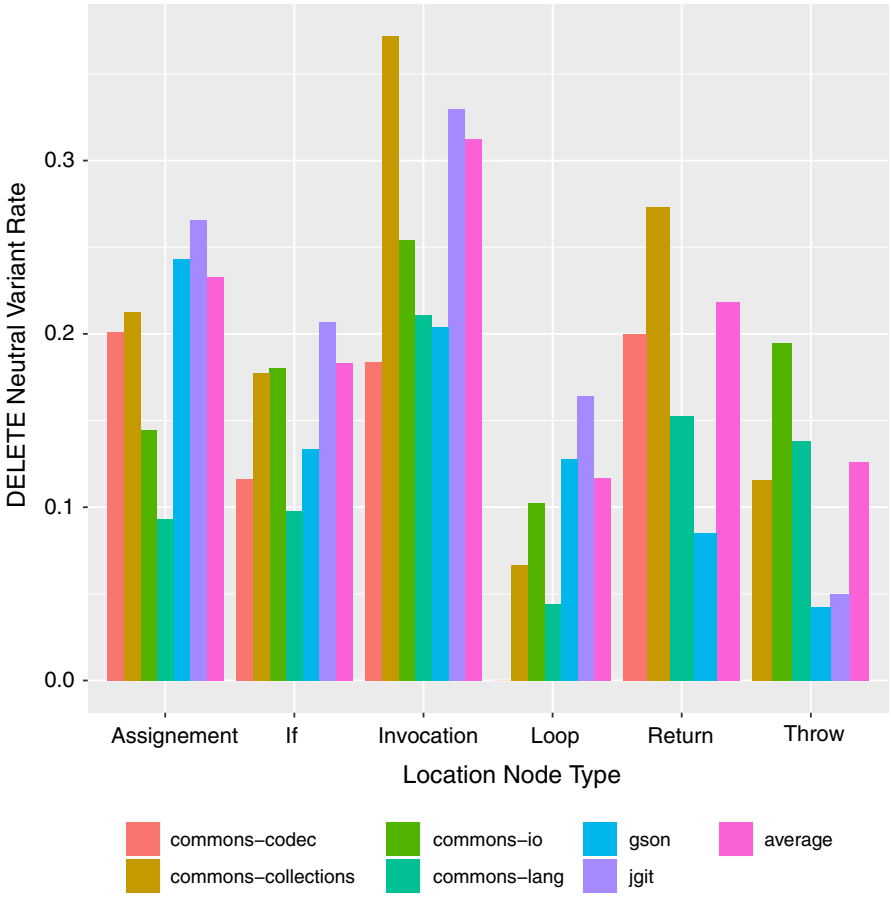

Fig. 7 Neutral variant rate of DELETE transformation in function of the type of the location

Table 4 Neutral variant rate of ADD, DELETE, and REPLACE by project and their 95% confidence interval

	ADD NVR	DELETE NVR	REPLACE NVR
commons-codec	45.51% ± 3.87	20.03% ± 2.91	10.55% ± 1.2
commons-collections	53.14% ± 1.14	23.63% ± 1.47	13.63% ± 0.39
commons-io	51.74% ± 1.68	19.35% ± 1.91	12.53% ± 0.6
commons-lang	42.45% ± 3.08	12.99% ± 1.72	11.05% ± 0.88
gson	48.04% ± 1.45	18.6% ± 2.24	16.74% ± 0.69
jgit	58.29% ± 1.68	26.7% ± 1.21	23.86% ± 0.75
Total	51.83% ± 0.69	22.48% ± 0.71	15.42% ± 0.26

were conducted per project, even if over all projects 102 trials were done.). Those nodes are usually used as shortcuts in the computation, hence removing them yields slower yet acceptable program variants; we discuss this in depth in the next section.

4.3.3 REPLACE

A REPLACE transformation can be seen as the combination of a DELETE and an ADD. Consequently, results are somewhat similar to the ones of ADD and DELETE. The neutral variant rate can be seen as the probability that the outcome of a transformation that compiles also passes the tests. This means that if ADD and DELETE transformations were independent for a given statement, the neutral variant rate for REPLACE should be close to the product of the two others. Yet, for each project (as shown in Table 4), the neutral variant rate for REPLACE is higher than this product, meaning that local neutral variant rate of ADD and DELETE are probably not independent.

We note two key phenomena. First, picking a transplant and a location that are method invocations is quite effective. This suggests the presence of alternative yet equivalent calls. This is similar to what is discussed in the next section and also by Carzaniga et al. [7]. It also appears that replacing an assignment by another one is efficient. Second, we observe a certain plasticity around "return" statements: some of them can be replaced by the statement surrounded by a "try" or a condition. This suggests the existence of similar statements in the neighborhood of the location, which perform additional checks.

Answer to RQ3: Generic, random program transformations can yield more than $23.30\% \pm 0.26$ neutral program variants, but not all code regions are equally prone to neutral variant synthesis. In particular, method invocations and variable assignments are more plastic than the rest of the code.

4.4 Role of plastic code regions

This section focuses on RQ4. Now, we are interested in understanding whether there is a difference in nature between the neutral variants and the variants that fail the test suite.

RQ 4 What roles do the code regions prone to neutral variant synthesis play in the program?

For each program, we selected neutral variant among extreme cases: those synthesized on locations covered by a single test case or synthesized on points covered by the highest number of test cases. By doing this, we are able to build a taxonomy of neutral variants.

This analysis is the result of more than two full weeks of work, where we have manually analyzed dozens of neutral variants. At a very coarse grain, before explaining them in detail, we distinguish three kinds of neutral variants: (1) *revealer neutral variants* indicate the presence of software plasticity in the code; (2) *fooler neutral variants* are named after Cohen's [11] counter-measures for security. (3) *buggy neutral variants* are made on locations that are poorly specified by the test suite, the transformation simply introduces a bug.

Revealer neutral variants take their denomination from the fact that they reveal something in the code that is implicit otherwise: *code plasticity*. Once those regions

are revealed, program transformation can target them, with a high confidence that the variant shall be neutral.

Fooler neutral variants are called like this in reference to the "garbage insertion" transformation proposed by Cohen [11]. These neutral variants add garbage code that can fool attackers who look for specific instruction sequences. To this extent, neutral variant synthesis can be seen as a realization of Cohen's transformation.

Buggy neutral variants are simply the degenerated and uninteresting by-products resulting from of weak test cases. We will not provide a taxonomy of buggy neutral variants.

In the following, we discuss categories of revealer and fooler neutral variants. For each category, we present a single archetypal example from the ones synthesized for this work (Table 2). Each example illustrates the difference in the original that produces a neutral variant. Examples come with a table that provides the values for the location features. A more complete set of examples is available online.[5]

Plastic specification Some program regions implement behavior which correctness is not binary. In other terms, there is no one single possible correct value, but rather several ones. We call such specification "plastic".

The regions of code implementing plastic specifications provide great opportunities for the synthesis of neutral variants, which transform the programs in many ways while maintaining valuable and correct-enough functionality.

One situation that we have encountered many times relates to the production of hash keys. Methods that produce these keys have a very plastic specification: they must return an integer value that can be used to identify an element. The only contract is that the function must be deterministic. Otherwise, there is no other constraint on the value of the hash key. Listing 1 illustrates an example of a neutral variant synthesized by removing a statement from a hash method (line 3). To us, the neutral variant still provides a perfectly valid functionality.

Listing 1 Delete a statement in `hash` (commons.collection)

```
1    int hash(final Object key) {
2      int h = key.hashCode();
3  -   h += (h << 9);
4      h ^= h >>> 14;
5      h += h << 4;
6      h ^= h >>> 10;
7      return h;}
```

#tc	transfo type	node type
422	del	var declaration

Optimization Some code is purely about optimization, which is an ideal plastic region. If one removes such code, the output is still exactly the same, only non-functional properties such as performance are impacted. Listing 2 shows an example of neutral variant that removes an optimization: at the end of the `if-block` (line 7), the original program stores the value of `buf` in `toString`, which allows to bypass the computation of `buf` next time `toString()` is

[5] https://github.com/castor-software/journey-paper-replication/tree/master/RQ4.

called; the neutral variant removes this part of the code, producing a potential performance degradation if the method is called intensively.

Listing 2 Delete a statement in `toString` (commons.lang)

```
1    String toString() {
2      String result = toString;
3      if (result == null) {
4        final StringBuilder buf = new StringBuilder(32);
5        [...] //...compute buf
6        result = buf.toString();
7  -     toString = result;
8      }
9      return result;}
```

#tc	transfo type	node type
2	del	stmt list

Code redundancy Sometimes, the very same computation is performed several times in the same program. For instance, two subsequent calls to `list.remove(o)`, even separated by other instructions are equivalent (as long as `list` and `o` do not change between). Program transformations naturally exploit this computation redundancy through the removal or replacement of these redundant statements. Replacement with a call to a side-effect free method also produces valid neutral variants.

Listing 3 displays an example of such a neutral variant (removing if-block at line 3). The statement `if (isEmpty(padStr)) padStr = SPACE;` assigns a value to `padStr`, then this variable is passed to methods `leftPad` and `rightPad`. Yet, each of these two methods include the exact same statement, which will eventually assign a value to `padStr`. So, the statement is redundant and can be removed from the original program, yielding a valid fooler neutral variant. Compared to neutral variants that remove some optimization, those neutral variants might perform better than the original program.

Listing 3 Delete in `center` (commons.lang)

```
1    String center(String str, final int size, String padStr) {
2      if (str == null || size <= 0) {return str;}
3  -   if (isEmpty(padStr)) {padStr = SPACE;}
4      [...]
5      str = leftPad(str, strLen + pads / 2, padStr);
6      str = rightPad(str, size, padStr);
7      return str;}
```

#tc	transfo type	node type
1	del	if

gram (replace at line 4), i.e., `((Object[]) object)[i]` has the same behavior as `Array.get(object, i)`, with completely different implementations.

Implementation redundancy It often happens that programs embed several different functions that provide the same service, in different ways. For example, there can

exist several versions of the same method with different sets of parameters, which can be used interchangeably by providing good parameter values. It is also possible to use libraries that provide this diversity of similar methods (as demonstrated by Carzaniga et al. [7]). Listing 4 illustrates the exploitation of such implementation redundancy inside the program (replace at line 4), i.e., `((Object[]) object)[i]` has the same behavior as `Array.get(object, i)`, with completely different implementations.

Listing 4 Replace in `get` (commons.collection)

```
1    Object get(final Object object, final int index) {
2      [...]
3      else if (object instanceof Object[]) {
4  -     return ((Object[]) object)[i];
5  +     try {
6  +       return Array.get(object, i);
7  +     } catch (final IllegalArgumentException ex) {
8  +       throw new IllegalArgumentException("Unsupported
9  +       object type: " + object.getClass().getName());
10 +     }
11     }
12     [...]
13   }
```

#tc	transfo type	node type
1	rep	return

Optional functionality In software, not all parts are of equal importance. Some parts represent the core functionality, other parts are about options and are not essential to the computation. Those optional parts are either not specified or the specification is of less importance. These are areas that can be safely removed or replaced while still producing useful variants. Listing 5 is an example of neutral variant that exploits such optional functionality. The neutral variant completely removes the body of the method, which is supposed to transform the type passed as parameter into an equivalent version that is serializable, and instead it returns the parameter. The neutral variant is covered by 624 different test cases, it is executed 6000 times and all executions complete successfully, and all assertions in the test cases are satisfied. This is an example of an advanced feature implemented in the core part of GSon that is not necessary to make the library run correctly.

Listing 5 Replace in `canonicalize` (GSon)

```
1    public static Type canonicalize(Type type) {
2  -   if (type instanceof Class) {
3  -     Class<?> c = (Class<?>) type;
4  -     return c.isArray() ? new
5  -     GenericArrayTypeImpl(canonicalize(c.getComponentType())) : c;
6  -   }
7  -   else
8  -   if (type instanceof ParameterizedType) {
9  -     ParameterizedType p = (ParameterizedType) type;
10 -     return new ParameterizedTypeImpl(p.getOwnerType(),
11 -     p.getRawType(), p.getActualTypeArguments());
12 -   }
13 -   else
14 -   if (type instanceof GenericArrayType) {
15 -     GenericArrayType g = (GenericArrayType) type;
16 -     return new GenericArrayTypeImpl(g.getGenericComponentType());
17 -   }
18 -   else
19 -   if (type instanceof WildcardType) {
20 -     WildcardType w = (WildcardType) type;
21 -     return new WildcardTypeImpl(w.getUpperBounds(),
22 -     w.getLowerBounds());
23 -   }
24 -   else {
25 -     return type;
26 -   }
27 +   return type;
28   }
```

#tc	transfo type	node type
623	rep	if

Listing 6 Add in `ensureCapacity` (commons.collection)

```
1    void ensureCapacity(final int newCapacity) {
2      final int oldCapacity = data.length;
3      if (newCapacity <= oldCapacity) {
4        return;
5      }
6      if (size == 0) {
7        threshold = calculateThreshold(newCapacity, loadFactor);
8        data = new HashEntry[newCapacity];
9      } else {
10       [...]
11     }
12 +   ensureCapacity(threshold);
13   }
```

#tc	transfo type	node type
8	add	invocation

Fooler neutral variants We have realized that a number of ADD and REPLACE transformations result in neutral variants which have more code than the original and where the additional code is harmless for the overall execution. These neutral variants act exactly as Cohen's "garbage insertion" strategy to fool malicious attackers, hence we call them fooler neutral variants.

We found multiple kinds of fooler neutral variants: some add branches in the code or redundant method calls or redundant sequences of method invocations. Some others reduce the legitimate input space through additional checks on input parameters. Listing 6 is an example of a fooler neutral variant, which adds a recursive call to `ensureCapacity()` (line 12). This could turn the method into an infinite recursion, except that in the additional recursive invocation, the value of the parameter is such that the condition of the first `if` statement always holds true and the method execution immediately stops. The additional invocation adds a harmless method call in the execution flow.

Discussion Let us now consider again the location features given for each neutral variant. Most neutral variants manually identified as buggy occur on locations covered by a single test case. In other words, the risk of synthesizing bad neutral variants increases when the number of test cases is low.

More interestingly, we realized that valid revealer and fooler neutral variants can be found both on points intensively tested and on weakly tested points. This confirms the intuition we expressed in the previous section: if a region is intrinsically plastic (has a plastic specification or is optional), the number of test cases barely matters, the only fact that the specification and the corresponding code region is plastic explains the fact that we can easily synthesize neutral variants.

Answer to RQ4: We have provided a first classification of plastic code regions according to the role this region plays in a program. The "revealers" indicate plastic code regions [30]. The "foolers" are useful in a protection setting [11]. Our manual analysis shows the variety of roles that code plays in a program. It uncovers the multitude of opportunities that exist to modify the execution of programs while maintaining a global, acceptable functionality.

4.5 Targeted transformations

RQ 5 Can program transformations target specific plastic code regions in order to increase their capacity of synthesizing neutral variants that exhibit behavioral variations?

For this question we design three novel, targeted program transformations: ADD METHOD INVOCATION that adds an invocation at the location, SWAP SUBTYPE that modifies the type of concrete objects that are passed to variables declared with an abstract type, and LOOP FLIP that reverses the order in which a loop iterates over a sequence of elements. These transformations refine the previous ADD, DELETE, REPLACE to target language constructs that are most likely plastic regions. Our intention is to design transformations that are more likely to produce variants that are syntactically correct, pass the same test suite as the original and exhibit a behavior that is different from the original. We assess the effectiveness of each targeted transformation with respect to:

- neutral variant rate, as defined in Definition 5
- behavior difference

We assess behavior difference by comparing the traces produced by the original and the neutral variant when running with the same input. For each targeted transformation, we select the relevant trace features that must be collected, in order to tune yajta (cf. Sect. 3.5). Then, the traces are aligned up until the first execution of the transformed region. If the traces diverge between a neutral variant and the original, we consider that the program transformation has, indeed, yield an observable behavioral difference. This reveals that (1) the transformation was performed on code that is not dead; (2) the compiler optimizations did not mask the effect of the transformations; and (3) two different executions can yield the same result. The assessment of behavioral differences through execution traces has proven useful in the search for patches fixing bugs in the field of automatic program repair [13].

4.5.1 ADD METHOD INVOCATION

The ADD METHOD INVOCATION transformation leverages the following observation: Fig. 6 indicates that the addition of "invocation" nodes is likely to produce neutral variants. We focus on invocations rather than loops or conditions to reduce the risk of synthesizing variants where the added code is not executed. We also exploit the good results obtained when adding "try" blocks.

4.5.2 The ADD METHOD INVOCATION transformation process

The transformation starts with the selection of a random location π. Then, it builds the set of methods that are accessible from π. A method is considered to be accessible if (1) the method is public, protected and in the same package as the class of π, or private and in the same class; (2) if π belongs to the body of a static method, the method called must be static. (3) if π does not belong to the body of a static method, the inserted invocation must either refer to a static method, or refer to a method member of the class of an object available in the context. (4) there exists a set of variables in the local context to fit the method's parameters. Let us notice that we prevent the method hosting π to be selected, as this would create recursive calls likely to produce an infinite loop.

Once a method m has been selected, we synthesize a transplant in the form of an invocation AST node to insert at the location. If the return type of m is not void, a public field is synthesized in the hosting class and the invocation result is assigned to this field. This additional rule aims at forcing the usage of the invocation's result and hence at preventing the compiler from considering the invocation as dead code and removing it [31]. The transplant is then wrapped into a "try-catch" block.

A formal definition of the transformation is provided in "Appendix".

4.5.3 Illustration of the ADD METHOD INVOCATION transformation

Listing 7 illustrates the addition of an invocation of `conditionC0(String, int)` before the `return` statement. Since `conditionC0` returns a boolean, a

public field of the same type is added to the `DoubleMetaphone` class to consume the result of the invocation.

Listing 7 Add method invocation in `DoubleMetaphone.java:882` (commons.codec)

```
882 + public boolean v6482819 = true;
883   private boolean isSilentStart(final String value) {
884     boolean result = false;
885     for (final String element : SILENT_START) {
886       if (value.startsWith(element)) {
887         result = true;
888         break;
889       }
890     }
891 +   try {
892 +     v6482819 = conditionC0(this.VOWELS, this.maxCodeLen);
893 +   } catch (Exception v4663426) {}
894     return result;
895   }
```

Figure 8 illustrates the juxtaposition of two dynamic call trees: the tree of the execution of `StringEncoderComparatorTest` on the original `isSilentStart` method and the tree when running the same test on the transformed method. Each node on the figure represents a method and each edge represents a method invocation. The temporal aspect of the execution is represented in two dimensions: method invocations go from top to bottom, and, if a method invokes several others, the calls on the left occur before those on the right. The nodes in grey represent calls the parts of the test execution that are common to both the original and the transformed program. Nodes in light green (and connected with dashed lines) represent the parts of the execution added with the transformation.

4.5.4 Searching the space of the ADD METHOD INVOCATION transformation

The size of the search space can be bound by the product of the number of statements in the targeted program and the number of methods it declares. In practice, we limit ourselves to methods for which we can pass parameters within the context of the location, which significantly reduces the size of the space. Yet, the space remains huge. Consequently, for experimental purposes, we limit our search to up to 10 different methods per location. If more than 10 methods can be invoked at the same point, we randomly select 10. As we performed sampling on the search space, in the rest of the subsection, we give NVR measures with their 95% confidence interval modelled following a binomial distribution.

4.5.5 Behavior diversity

To assess the behavioral variations introduced by the addition of a method invocation, we use **yajta** to trace the number of times each method in the program invokes

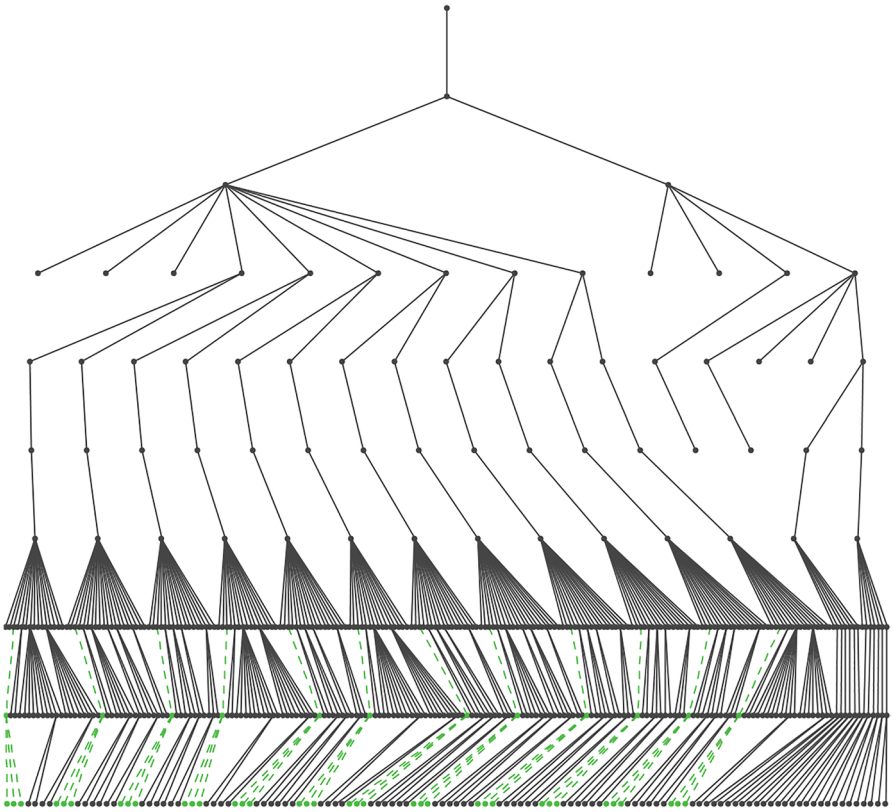

Fig. 8 Impact of a modification on StringEncoderComparatorTest call tree

any other method. This observation produces a $N \times N$ matrix, where N is the number of methods executed when running the test suite. The comparison of the matrix produced on the original program and the one produced on the variant reveals if it is, indeed, possible to observe additional method invocations (i.e., additional behavior) at runtime.

Table 5 shows an excerpt of the trace when running `StringEncoderComparatorTest`. Each line records the number of times a method has invoked the methods mentioned in the column header. The results recorded during the execution of the test on the original program appear in black, while the new calls, occurring as a result of the transformation, appear in green. We observe that the transformed method (`isSilentStart`) is called 12 times by `doubleMetaphone` during the test run, on the original program. The program transformation adds an invocation to `conditionC0` in `isSilentStart`. This results in 12 invocations of `conditionC0`, as well as 12 times more invocations to all the methods invoked by `conditionC0`. These can be observed in Fig. 8 as 12 subtrees of one node calling 3 other appear in green.

Table 5 Call Matrix of the execution of `StringEncoderComparatorTest` when adding a call to `conditionC0` in `isSilentStart` (in DoubleMetaphone)

	...	isSilentStart(String)	conditionC0(String, int)	contains(String, int, int, String[])	charAt(String, int))	isVowel(char)	...
...	...	...	...	...	...	...	...
isSilentStart(String)	...	0	0 +12	0	0	0	...
conditionC0(String, int)	...	0	0	0 +12	0 +12	0 +12	...
doubleMetaphone(String, boolean)	...	12 +0	0	0	2 +0	0	...
...	...	...	...	...	...	...	...

Table 6 Neutral variant rate of ADD METHOD INVOCATION

	#Locs	#Compiles	#NV	NVR
commons-codec	1722	17,650	11,150	63.17% ± 0.71
commons-collections	7027	40,333	26,150	64.84% ± 0.47
commons-io	1608	10,009	7413	74.06% ± 0.86
commons-lang	4287	129,593	86,452	66.71% ± 0.26
gson	2460	32,932	18,215	55.31% ± 0.54
jgit	12,822	28,582	22,364	78.25% ± 0.48
Total	29,926	259,099	171,744	66.29% ± 0.18

4.5.6 Empirical results for the ADD METHOD INVOCATION transformation

Table 6 displays the results per study object: (#Locs) number of locations for which transformations were attempted, number of times we performed the ADD METHOD INVOCATION transformation and produced a compilable variant (# Compile); number of transformations that yield a neutral variants (# NV); and the neutral variant rate (NVR). Overall, 66.29% of the program transformations yield a program variant that compiles and passes the suite, which corresponds to 171,744 neutral variants in total.

The first key observation is that method invocations are plastic regions, regardless of the original program. The second observation is that the targeted program transformation is significantly (p value < 0.001 with a `Wilcoxon rank sum test`) more effective than a random invocation addition to synthesize neutral variants: 66.29% on average instead of the 45% neutral variant rate of the ADD transformation presented in Fig. 6 when inserting method invocation.

Several factors contribute to this successful synthesis of neutral variants. First, the transformation selects the methods to be added, ensuring that it is possible to get valid parameter values in the context of the location. This design decision can favor repeating an invocation that already exists in the method that hosts the location. If the method is idempotent, the trace changes with no side effect. Second, the additional invocation is wrapped into "try" blocks. This may also lead to the compilation of invocations that quickly throw an exception and therefore, do not cause any state change. In general, the addition of invocations to idempotent (i.e. methods that have no additional effect if they are called more than once with the same input parameters) or pure methods (i.e. method with no externally observable side effect [32]) can make the insertion benign.

In Table 7 we provide the cumulative neutral variant rates, with respect to the type of method in which the location is selected (location (Loc) in static or non-static method) and with respect to the type of transplant (invoke a method that inside the same class as the location or that is external to that class). In this table, we observe a significant (p value < 0.001 with a `Wilcoxon rank sum test`) difference between the two types of locations: locations in static methods are more plastic (87.06%) that in non static ones (63.85%). We hypothesize that this comes from the fact that in the case of a location inside a static method, the additional invocation can only be towards a static method. Increased neutral

Table 7 Neutral variant rate of ADD METHOD INVOCATION depending on Transplant and Location type

	Internal Transplant	External Transplant	Total
Static Loc	85.92% ± 0.62	87.96% ± 0.52	87.06% ± 0.40
Non static Loc	62.52% ± 0.20	80.89% ± 0.59	63.85% ± 0.20
Total	63.75% ± 0.20	84.25% ± 0.40	66.29% ± 0.18

variant rate in this case could come from the fact the proportion of pure methods is higher among static methods than among regular methods.

We also observe more successful transformations when the transplant is selected outside the class that hosts the location (84.25% instead of 63.75%, p value < 0.001 with a `Wilcoxon rank sum test`). We hypothesize that methods invoked in the same class as the location are likely to be non-pure methods. The transformation selects invocations to methods for which the context of the location can provide values to pass as parameters. This means that most of the methods inside the same class can be invoked, whereas in the case of external methods this tends to select methods with no parameter or methods that have only parameters of primitive data types. We hypothesize that this difference in the selection of candidate methods increases the chance to have more pure methods among external than among internal method invocations.

4.5.7 SWAP SUBTYPE

The results of the REPLACE transformation showed that targeting assignment statements yields more neutral variants than on other types of AST nodes. In this section, we introduce a new transformation that refines REPLACE on "Assignment", leveraging Java interfaces. A common practice in Java consists of declaring a variable typed with an interface. When a developer adopts this practice, she indicates that any concrete object that implements the interface can be assigned to this variable. The existing diversity of available types sharing an interface can be leveraged to fuel our search for neutral variants.

4.5.8 The SWAP SUBTYPE transformation process

This program transformation operates on assignment statements that pass a new concrete object to a variable typed with an interface. The transformation replaces the constructor called in such assignments by one of a class implementing the same interface. In the following experiments we have implemented this transformation for classes and interfaces of Java collections.

4.5.9 Illustration of the SWAP SUBTYPE transformation

Listing 8 shows an example of a SWAP SUBTYPE transformation, while Fig. 9 illustrates its impact on the dynamic call tree of one test. Nodes in light teal are

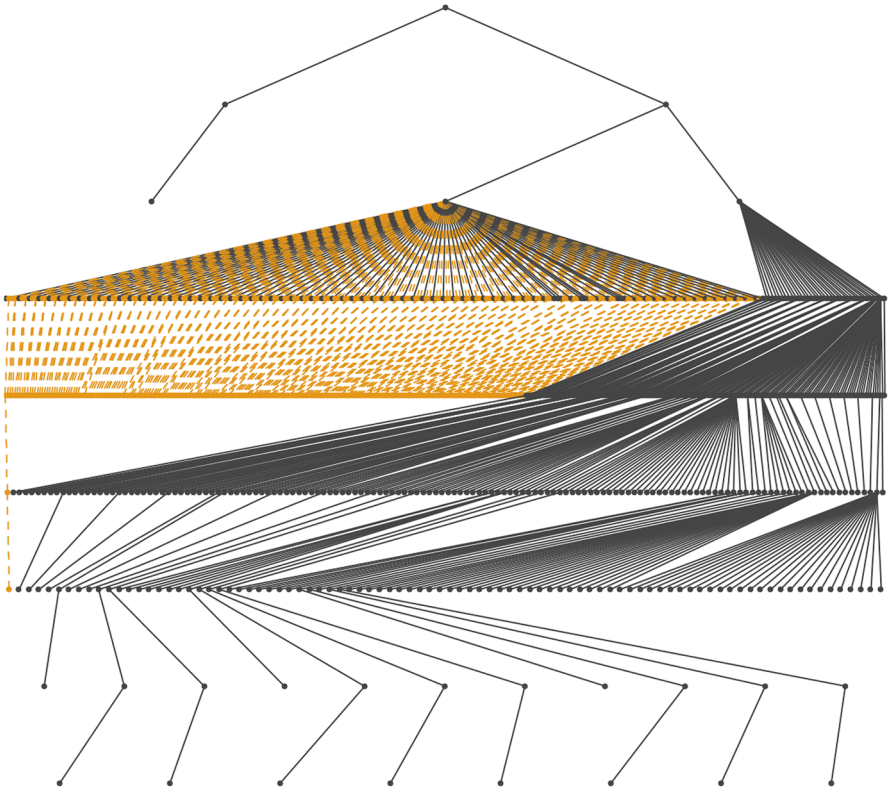

Fig. 9 Impact of a modification on the call tree of one execution of `PhoneticEngineTest.testEncode()`

method invocations from *org.apache.commons.collections*4 which were not present before. (They replace previous calls to the Java standard library).

Listing 8 SwapSubType in `Lang.java:130` (commons.codec)

```
130     public static Lang loadFromResource(final String
            languageRulesResourceName, final Languages languages) {
131 -     final List<LangRule> rules = new ArrayList<LangRule>();
132 +     final List<LangRule> rules = new
            org.apache.commons.collections4.list.NodeCachingLinkedList<LangRule>();
133         [...]
134     }
```

4.5.10 Searching the space of SWAP SUBTYPE transformations

The search space here is composed of all statements that assign a new concrete object to a variable which type is a collection (see "SwapSubtype" section in

Appendix for the actual list). This space is small enough to be explored exhaustively. We target 16 interfaces, which are implemented by 50 classes (some of which implement several interfaces) from 3 different libraries (`java.util`, `org.apache.commons.collections` and `net.sf.trove4j`). The complete list of interfaces, and their concrete classes, targeted by this transformation is available in the replication repository. This transformation is similar to what Manotas et al. [24] have implemented in their framework SEEDS. While the choice of a concrete collection might be a long planned decision for performance reasons, we believe that in many cases the choice is made by default.

4.5.11 Behavior diversity

To observe the changes introduced by the SWAP SUBTYPE transformation, we use **yajta** to trace both the methods defined in the classes of the program that is transformed and all the methods in collection classes that are involved in the transformation (the ones at the location and the ones in the transplants). The trace comparison procedure is the same as for the ADD METHOD INVOCATION transformation.

4.5.12 Empirical results for the SWAP SUBTYPE transformation

Table 8 presents the results of the SWAP SUBTYPE transformation on each project of our sample. In total, we synthesized 4909 variants that compiled on 339 different locations (i.e. collection assignment to a variable typed as an interface for which at least one transformation yields a variant that compiles). Out of the 4909 variants that compile correctly, 2860 are neutral variants. This represents a global 58.26% neutral variant rate. We notice that the SWAP SUBTYPE transformation yields more than 80% neutral variants for 4 projects. Yet, for `jgit` and `commons-collections`, the neutral variant rate falls to 46.89% and 60.98% respectively. Overall this represents a geometric mean of 74%.

Table 8 Neutral variant rate of SWAP SUBTYPE

	#Loc	#Compile	#NV	NVR (%)
commons-codec	21	186	164	88.17
commons-collections	68	738	450	60.98
commons-io	16	183	177	96.72
commons-lang	41	544	445	81.80
gson	17	266	221	83.08
jgit	190	2992	1403	46.89
Total	339	4909	2860	58.26

A major reason for the lower neutral variant rate on `commons-collections` is the use of inner classes that implement the `Collection` interface. This happens to create classes that mix the contract of the `Collection` interface with the contract of the class inside which the `Collection` interface implementation is defined. For example, the class `MultiValueMap$Values` implements the iterator of the `Collection` interface inside `MultiValueMap`. Listing 9 shows an instantiation of `MultiValueMap$Values` that was used as a location for the SWAP SUBTYPE transformation. The original program assigns a `MultiValueMap$Values` to `valuesView`. This means that subsequent calls to `MultiValueMap$Values.iterator()` return the values that are stored in the field `map`. Now, since `vs` is a of type `Collection`, the SWAP SUBTYPE transformation assumes that it can assign it any object typed with an implementation of `Collection`, e.g. `LinkedList` in this example. Yet, because a call to `iterator()` on an instance of `LinkedList` only iterate over elements that have been added to the instance, all `MultiValueMap$Values.iterator()` calls return empty iterators which leads to failing tests. Such situations occurred for 113 variants, 0 of which are neutral.

Listing 9 SwapSubType in `MultiValueMap.java:326` (commons.codec)

```
326       @Override
327       @SuppressWarnings("unchecked")
328       public Collection<Object> values() {
329           final Collection<V> vs = valuesView;
330   -       return (Collection<Object>) (vs != null ? vs : (valuesView = new
          Values()));
331   +       return (Collection<Object>) (vs != null ? vs : (valuesView = new
          LinkedList<V>()));
332       }
```

While the number of candidates to be targeted by this transformation is lower than for other transformations, SWAP SUBTYPE affects all subsequent invocations that target the modified variable. Therefore, the program transformation impacts the generated variant in a more profound way than other transformations. This effect is well illustrated by Fig. 9.

In theory, it is possible to swap any valid subtype of an interface when assigning a concrete object to a variable typed with the interface, and this with no effect on the functionality. This property is a direct consequence of the fact that any requirement on the type of a variable should be expressed in the interface. In other words, SWAP SUBTYPE should be a sound preserving transformation. Indeed, we observe that there exist at least one neutral variant for 71% of the 339 locations targeted by the SWAP SUBTYPE transformation. However, in practice we observe that is not always the case, and SWAP SUBTYPE is, indeed, a program transformation: only 58% of the transformations actually yield a neutral variant.

Listing 10 Altering an `ordered loop` in ReflectiveTypeAdapterFactory:140 (gson)

```
140       List<String> fieldNames = getFieldNames(field);
141       BoundField previous = null;
142   -   Map<String, BoundField> result = new LinkedHashMap<String, BoundField>();
143   +   Map<String, BoundField> result = new HashMap<String, BoundField>();
144       [...]
145        for (int i = 0; i < fieldNames.size(); ++i) {
146         String name = fieldNames.get(i);
147         if (i != 0) serialize = false; // only serialize the
              default name
148         BoundField boundField = createBoundField(context, field,
              name,
149         TypeToken.get(fieldType), serialize, deserialize);
150         BoundField replaced = result.put(name, boundField);
151         if (previous == null) previous = replaced;
152       }
```

Listing 10 illustrates an example where the SWAP SUBTYPE transformation fails at producing a neutral variant. Here, the concrete type in the original program is `LinkedHashMap`. This specific implementation of the `Map` interface keeps the entries in the order of insertion. When the `for` loop iterates through the `fieldNames` list, the `result` map is filled such that the elements in `map` are stored in the same order as the elements in `fieldNames`. Now, when the *swap subtypes* transformation assigns a `HashMap` object to `result` instead of a `LinkedHashMap`, the elements of `result` are ordered with respect to their hash value instead of keeping the order of insertion of `fieldNames`. Consequently, subsequent methods that expect a specific order in `result` fail because of this change.

It is important to notice that when we replace `LinkedHashMap` by `org.apache.commons.collections4.map.LinkedMap` in Listing 10, the corresponding variant is neutral, since the substitute type satisfies the required invariant: elements are kept in order of insertion. More generally, we can say that this zone is plastic, modulo this type invariant.

4.5.13 LOOP FLIP

Swapping instructions is a state of the art transformation used by Schulte et al. [34] or in a sound way for obfuscation [38]. Here we explore a targeted swap transformation, which reverses the order of iterations in loops.

4.5.14 The LOOP FLIP transformation process

We propose a program transformation that reverses the order in which `for` loops iterate over a set of elements. It targets counted loops, i.e., loops for which we can identify a loop counter variable that is initialized with a specific value and which is increased or decreased at each iteration until it satisfies a condition. The transformation does not necessarily expect a well-behaved counted loop. The transformation makes the loop run the same iterations as the original loop, but for loop index values in reverse order. To achieve this, we need to identify the initial value, the step, and the last value. Listing 12 shows such an example for a simple case. The loop counter

is the variable i, its initial value is 0, the step is +1, so the last value is straightforward to determine ($srcArgs.length - 1$). In this example, the transformation replaces the original loop with one starting from the last value, with a step of −1 and ending when the loop counter reaches the initial value.

The example of Listing 11 is a non-normalized loop that we still handle with LOOP FLIP. The variable i is still the loop counter. Its first value is 0, and its last value is 28 as 32 is not reachable. For the general case, the last value is the last multiple of *step* smaller than the difference between the upper bound and the starting value. Yet, as we only transform the code in a static way, this expression is directly inserted in the initialization of the loop counter. More implementation details are given in the replication repository.[6]

Listing 11 Loopflip in `MemberUtils.java:115` (commons.lang)

```
115 -    for (int i = 0; i < srcArgs.length; i++) {
116 +    for (int i = srcArgs.length-1; i >= 0; i--) {
```

Listing 12 Loopflip in `UnixCrypt.java:87` (commons.codec)

```
87  -    for (int i = 0; i < 32; i += 4) {
88  +    for (int i = 32 - (((32 - 0) % 4) == 0 ? 4 : (32 - 0) % 4); i >= 0; i -=
         4) {
```

4.5.15 Illustration of the LOOP FLIP transformation

In order to illustrate how test execution may be affected by this transformation, Listing 11 details a transformation, a test that cover the transformed code and its execution trace.

Listing 13 Loopflip in `BinaryCodec.java:108,123` (commons.lang)

```
108   public static byte[] toAsciiBytes(final byte[] raw) {
109       if (isEmpty(raw)) {
110           return EMPTY_BYTE_ARRAY;
111       }
112       final byte[] l_ascii = new byte[(raw.length) << 3];
113       for (int ii = 0, jj = (l_ascii.length) - 1 ; ii <
              (raw.length) ; ii++ , jj -= 8) {
114   -   for (int bits = 0 ; bits < (BITS.length) ; ++bits) {
115   +   for (int bits = (BITS.length - 1) ; bits >= 0 ; --bits) {
116               if (((raw[ii]) & (BITS[bits])) == 0) {
117   A               l_ascii[(jj - bits)] = '0';
118               } else {
119   B               l_ascii[(jj - bits)] = '1';
120               }
121           }
122       }
123       return l_ascii;
124   }
```

[6] https://github.com/castor-software/journey-paper-replication/tree/master/RQ5.

Listing 13 shows an example where the LOOP FLIP transformation yields a neutral variants of the `BinaryCodec.toAsciiBytes()` method. This method returns an array of `bytes`, which is filled inside the loop that we transform. The variant is neutral because the array is filled with values which do not depend on the iteration order of the loop, but only on the value of the loop index. We can think of this as filling the table with a set of numbered operations, where the order in which the operations are performed does not matter. Consequently the `l_ascii` array contains exactly the same thing, whatever the iteration order of the `for` loop in line 114.

Listing 14 Call to toAsciiBytes in `BinaryCodecTest.java:706,709` (commons.codec)

```
706     bits = new byte[1];
707     bits[0] = BIT_0; //0b00010111
708     l_encoded = new String(BinaryCodec.toAsciiBytes(bits));
709     assertEquals("00010111", l_encoded);
```

Listing 14 shows an excerpt of the test case that specifies the behavior of `BinaryCodec.toAsciiBytes()`. It calls the method with the binary value 00010111 as parameter and assesses that the return value is the array of `bytes` that encodes the String "00010111". Figures 10 and 11 show the execution of both the original and transformed method in that context. Round nodes correspond to method calls, squared ones correspond to branches in the order where they are called (from left to right). The branch highlighted in orange (dashed line) corresponds to the

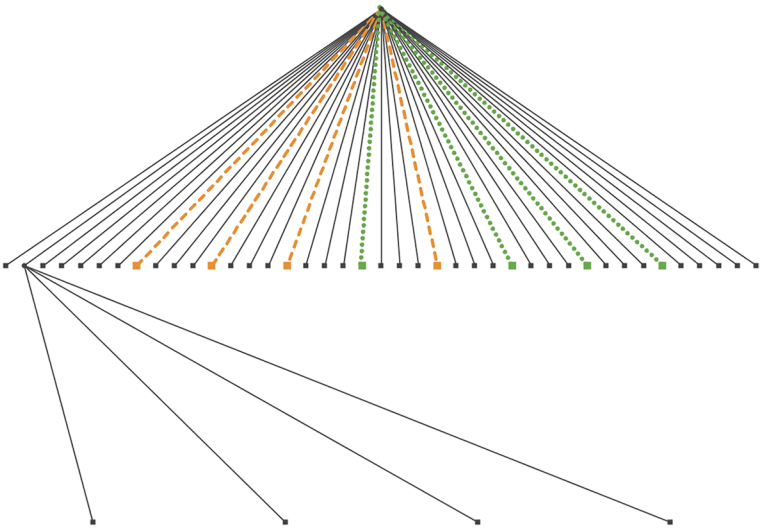

Fig. 10 Original test execution of BinaryCodec.toAsciiBytes. (Note the pattern AAABABBB). Round nodes correspond to method calls, squared ones correspond to branches in the order where they are called (from left to right). The branch highlighted in orange (dashed line) corresponds to the line A (in the same color in Listing 13). The branch highlighted in green (dotted line) corresponds to the line B.

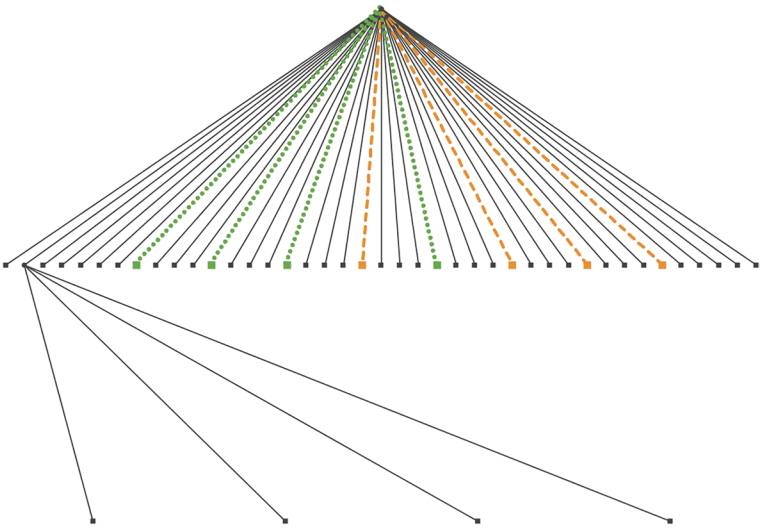

Fig. 11 Transformed test execution of BinaryCodec.toAsciiBytes. (Note the pattern BBBABAAA). Round nodes correspond to method calls, squared ones correspond to branches in the order where they are called (from left to right). The branch highlighted in orange (dashed line) corresponds to the line A (in the same color in Listing 13). The branch highlighted in green (dotted line) corresponds to the line B.

line A (in the same color in Listing 13). The branch highlighted in green (dotted line) corresponds to the line B. We can observe that the execution order is indeed reversed.

4.5.16 Searching the space of LOOP FLIP transformations

In the case of this transformation, the search space is composed of for-loops based on an integer index. Since it is fairly small, we exhaustively explore it.

4.5.17 Behavior diversity

The observation of branch executions would not be enough to systematically detect behavioral differences caused by this transformation for every case. Indeed for a loop whose body is composed of a single branch, branches executed do not depend on the index variable, therefore, branch observation would fail to detect differences. Thus the simplest observation method is to insert a probe at the beginning of the transformed loop to trace the value of the loop index.

4.5.18 Empirical results for the LOOP FLIP transformation

Table 9 summarizes the results of the LOOP FLIP experiments. We observe that this program transformation is very effective at synthesizing neutral variants. In total, we synthesized 479 neutral variants out of 656 variants, that compiled, targeting each a different `for` loop, which corresponds to a global neutral variant

Table 9 Neutral variant rate of LOOP FLIP

	#Locs	#Trials	#NV	NVR (%)
commons-codec	42	42	31	73
commons-collections	61	61	56	92
commons-io	35	35	24	69
commons-lang	227	227	146	64
gson	17	17	11	65
jgit	274	274	211	77
Total	586	586	427	73

rate of 73%. This neutral variant rate varies from 64 to 92% in `commons-collections`. This is significantly higher than any of the random program transformations analyzed previously.

The high neutral variant rate of LOOP FLIP can be explained by the fact that in many cases this transformation processes loops in which there are no loop-carried dependencies [14] (e.g., Listing 13). Meanwhile we can also note that both the number of candidates and the neutral variant rate vary widely from one project to another. This can be explained by different usages of loops in different projects. For example, if a project uses `forEach` loops more often than `for` loops, then the number of candidates for our transformation decreases. Also, `for` loops are used for different purposes: in some cases this control structure is used to apply the same computation to n elements that are independent of each other, whereas in other cases it is used to sequence of computations in which each action depends on the previous one. In the former case, the order of the loop iteration does not matter, while in the latter case, flipping loop order is very likely to modify the global behavior.

Listing 15 Altering an `ordered loop` in ReflectiveTypeAdapterFactory:140 (gson)

```
140       List<String> fieldNames = getFieldNames(field);
141       BoundField previous = null;
142       Map<String, BoundField> result = new LinkedHashMap<String,
              BoundField>();
143       [...]
144   -   for (int i = 0; i < fieldNames.size(); ++i) {
145   +   for (int i = (fieldNames.size()) - 1; i >= 0; --i) {
146         String name = fieldNames.get(i);
147         if (i != 0) serialize = false; // only serialize the
                default name
148         BoundField boundField = createBoundField(context, field,
                name,
149         TypeToken.get(fieldType), serialize, deserialize);
150         BoundField replaced = result.put(name, boundField);
151         if (previous == null) previous = replaced;
152       }
```

For example, Listing 15 illustrates a LOOP FLIP transformation that yields a variant that is not neutral. This case is similar to the one discussed on Listing 10:

when changing the iteration order, the `result` map is filled in a different order than in the original case. Consequently, the behavior of the method changes, which does not correspond to the expectation of the callers and eventually fails some test cases.

4.5.19 Research Question 5 Conclusion

In this section we have leveraged the observations made with generic, random program transformations, in order to design three new transformations that target code regions which are very likely plastic. When designing these transformations, we also increased the amount of static analysis performed by the transformation, leveraging the strong type system of Java. Overall, these design decisions aim at focusing the search on spaces of program variants with high densities of neutral variants. The results confirm these higher densities, with neutral variant rates of 66% (ADD METHOD INVOCATION), 58% (SWAP SUBTYPE), 73% (LOOP FLIP) that are significantly higher than the rates with generic, random transformations 23% overall (p value < 0.001 for each of the three `Wilcoxon rank sum test`).

Beyond the results and observations made with these three transformations, the experiments reported here are very encouraging to explore the 'grey' zone that exists between sound and semantic preserving transformations at one extreme and random, generic highly program transformations on the other extreme. We believe that in-depth knowledge about the nature of plastic code regions, combined with static code analysis is essential to design transformations that explore spaces of program variants that are behaviorally diverse, while limiting the amount of resources required to explore these spaces.

Answer to RQ5 Program transformations targeted at specific plastic code regions are significantly more effective than random transformations at synthesizing program variants, which exhibit visible behavior diversity and are equivalent modulo test suite. This RQ has explored three targeted program transformations that yield 66%, 58%, 73% neutral variants.

5 Discussion

Our journey among the different factors that influence the synthesis of *neutral program variants* has shed the light on several key findings. We have observed that many neutral variants result from the very specific combinations of one program transformation on one specific type of language structure. For example, the DELETE transformation in "invocation" nodes is surprisingly effective at synthesizing neutral variants, while it performs very poorly on "loop". Similarly, the ADD transformation is very effective with "try" nodes, but is very bad with "return".

These observations are novel and very interesting to design program transformations in future work. Yet, we believe that the most intriguing findings of our work relate to regions of the code that are plastic by nature, and not, by chance, because of one specific transformation.

The functional contract of a code region is what ultimately determines if a variant of that region is neutral or not. Such a contract defines a set of properties about the inputs and outputs of the code region, as well as state invariants for that region. Consequently, a contract can be more or less restrictive on the behaviors that implement the contract. Our empirical inquiry of program transformations has revealed that some contracts define loose expectations about the behavior of a code region. In turn, these code regions are more plastic than other parts.

Here are three examples of code regions with loose contracts:

- the contract of a `hash function` (e.g. the one of Listing 1) loosely specifies the returned value:[7] it only enforces the result to be a deterministic integer only depending on information used in `equals`. In addition, a weak requirement is that this method should avoid collision. This means any transformation, which side effect is to change the return in a deterministic way, yields a variant that fulfills the contract, even if changing the likelihood of collision impacts performance.
- the contract over some data ordering. For example, data structures that do not impose an order on their elements, or loops with no loop-carried dependence are code regions that have a loose contract. These regions tolerate many types of transformations that change order, for example, LOOP FLIP or SWAP SUBTYPE in case where an ordered collection is replaced by another a non-ordered one.
- optional functionalities (e.g., optimization code). The elective nature of these code regions make them naturally loosely specified. These functionalities are called by other functions, and the functional contract is defined on these other functions, not on the optional ones. All transformations that remove or modify the optional functionality produce program variant that is very likely to satisfy the contract.

In this work, we have used unit test suites as proxies for functional contracts. As discussed in Sect. 4.4, this might lead to false positives (variants considered neutral modulo the test suite, but that happen to be buggy variants). Yet, in many cases, this also allowed us to spot *inherently plastic* code regions that are prone to several program transformations, which can synthesize more neutral variants.

6 Threats to validity

We performed a large scale experiment in a relatively unexplored domain: the characterization of plastic code regions. We now present the threats to validity.

While we aim at analyzing code plasticity, we actually measure the rate of neutral variants produced by specific program transformations. This can raise a threat to the construct validity of our study, with respect to two concerns: (1) the limitation

[7] Oracle's documentation (Java 8): https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--.

of plasticity to a given transformation, (2) the confinement of changes only to the source code but not to the behavior. We mitigate the first concern through the manual analysis in our answer to RQ4 that emphasizes the presence of real code plasticity and not only plasticity related to a given transformation. To mitigate the second, we analyzed, in RQ5's answer, the execution traces proving actual, observable differences in execution.

Our findings might not generalize to all types of applications. Depending on the type of applications and the quality of their test suite, the obtained results could change. To minimize the impact of this threat, we selected open source frameworks and libraries because of their popularity, their longevity and the very high quality of their test suites. In addition, we provided an explicit analysis of the impact of tests on the neutral variant rate of transformations in Sect. 4.2.

Finally, our large scale experiments rely on a complex tool chain, which integrates code transformation, instrumentation, trace analysis and statistical analysis. We also rely on the Grid5000 grid infrastructure to run millions of transformations. We did extensive testing of our code transformation infrastructure, built on top of the Spoon framework that has been developed, tested and maintained for more than 10 years. However, as for any large scale experimental infrastructure, there are surely bugs in this software. We hope that they only change marginal quantitative results, and not the qualitative essence of our findings. Our infrastructure is publicly available on Github.[8]

7 Related work

Our work is related to seminal work analyzing the capacity of software to yield useful variants under program transformations. It is also related to work that exploits program transformations (either random or targeted) to improve software. Here, we discuss the key work in these areas, as well as the novelty of our work.

7.1 Plasticity of software

The work on *mutational robustness* by Schulte et al. [34] is a key inspiration for our own work. These authors explore the ability of software to be transformed under random copy, deletion and swap of AST nodes. Their experiments on 22 small to medium C programs (30 to 60 K lines of code) show that 30 % of the transformations yield variants that are equivalent to the original, modulo the test suite. They call this property of software *mutational robustness*. More recently, this research group demonstrate that the interaction of several neutral mutations can lead a program to exhibit new positive behavior such as passing an additional test. They call this phenomenon *positive epistasis* [29]. Other work has since confirmed the existence of mutational robustness [17, 22]

[8] https://github.com/castor-software/journey-paper-replication.

Our RQ1 can be considered as a conceptual replication [36] of the work by Schulte and colleague. Our results mitigate two threats to the validity of Schulte's results: our methodology mitigates internal threats, by using another tool to perform program transformations, and our experiment mitigates external threats by transforming Java programs (instead of C). Similarly to Schulte, we conclude "that mutational robustness is an inherent property of software". Yet, our study also provides completely novel insights about the language constructs and the code areas that support mutational robustness (we call them *plastic code regions*) and about the effectiveness of targeted transformations to maximize the synthesis of neutral variants.

Recently, Danglot et al. [12] have also explored the capacity of software and absorbing state transformations. They explore *correctness attraction*: the extent to which programs can still produce correct results under runtime state perturbations. In that study the authors rely on a perfect oracle to asses the correctness of outputs, and they observe that many perturbations do not break the correctness in ten subject programs. Our work also shows that program variants can have different traces and still deliver equivalent results (modulo the test suite). Yet, we rely on different transformations and we analyze in-depth the nature of the code regions that can yield neutral variants.

Our work extends the body of knowledge about forgiving code regions [30]. In particular, we find regions characterized by "plastic specifications", i.e. regions which are governed by a very open yet strong contract. For instance, the only correctness contract of a hashing function is to be deterministic. On the one hand this is a strong contract. On the other hand, this is very open: many variants of a hashing function are valid, and consequently, many modifications in the code result in valid hashing functions.

Some recent work investigate a specific form of software plasticity, referred to as *redundancy* [8, 16, 37]. This work consider that a code fragment is redundant with another fragment, in a specific context, if in that context, both fragments lead a program from a given state to an equivalent one through a different series of intermediate state. This is very close to neutral variants, which have diverse visible behavior and yet satisfy the same properties as assessed by the test suite. The key difference between our work is that we investigate program transformations to synthesize neutral variants, i.e. increase redundancy, whereas they analyze redundancy that naturally occurs in software systems.

7.2 Exploiting software plasticity

Genetic improvement [28] is an area of search-based software engineering [18], which consists in automatically and incrementally generating variants of an existing program in order to either improve non-functional properties such as resource consumption or execution time, or functional ones (e.g. automatic repair). All variants should pass the test suite of the original program. Existing work in this domain rely on random program transformations to search for program variants: Schulte et al. [33] exploit mutational robustness to reduce energy consumption; Langdon and Petke [21] add, delete, replace lines in C, C++, CUDA program sources to improve

performance; Cody-Kenny et al. [10] add, delete, replace AST nodes, to profile program performance; López et al. [23] explore program mutations to optimize source code. Manotas and al. [24] replace Java collections to optimize energy consumption. All this work leverage the existence of code plasticity, and the performance of the search process can be improved with targeted program transformations. In particular, our results with the SWAP SUBTYPE transformation, show that changing library is very effective to generate neutral variants, and this transformation is a key enabler to improve performance [3].

Software diversification [5] is the field concerned with the automatic synthesis of program variants for dependability. Existing work in this area also intensively exploit software plasticity and program transformations: Feldt [15] was among the first to use genetic programming to generate multiple versions of a program to have failure diversity; we relied on random transformations to synthesize diverse implementations of Java programs [1, 4]; recent work on composite diversification [38], investigate the opportunity to combine multiple security oriented transformation techniques. This work can benefit from our findings about targeted program transformations, which introduce important behavior changes (in particular the SWAP SUBTYPE transformation), while maximizing the chances of preserving the core functionality.

Shacham et al. [35] and, more recently, Basios et al. [3] investigate source code transformations to replace libraries and data structures, in a similar was as the SWAP SUBTYPE transformation. This corroborates the idea of a certain plasticity around these data structures, and the notion of interface.

8 Conclusion

The existence of neutral program variants and the ability to generate large quantities of such variants are essential foundations for automatic software improvement. Our work contributes to these foundations with novel empirical facts about neutral variants and with actionable transformations to synthesize such variants. Our empirical analysis explores the space of neutral variants of Java programs, focusing on 6 large open source projects, from different domains. We generated 98,225 variants that compile for these projects, through program transformations, and 23,445 were neutral variants, i.e., more than 20% of the variants run correctly and pass the same test suite as the original. A detailed analysis of these neutral variants revealed that some language constructs are more likely to be plastic than others to the synthesis of neutral variants (for example, method invocations) and also that some code regions have specific roles that make them plastic (for example, optimization code).

The actionable contribution of our work comes in the form of three novel program transformations for Java programs. We have designed these transformations to target specific code regions that appear more prone to neutral variant synthesis. Our experiments show that these transformations perform significantly better than generic ones: 60% (ADD METHOD INVOCATION), 58% (SWAP SUBTYPE), 73% (LOOP FLIP) instead of 23,9%.

One key insight from the series of experiments reported in this work is that some code regions are *inherently* plastic. These code regions are naturally prone to

behavioral variations that preserve the global functionality. These regions include code that has a plastic specification (e.g., hash function); optional functionality (e.g., optimization code) or regions that can be naturally reordered (e.g., loops with no loop-carried dependence). In our future work, we wish to leverage this insight about the deep nature of large programs to develop techniques that can generate vast amounts of software diversity for obfuscation [19] and moving target defenses [26].

Acknowledgements We would like to thank Amine Benelallam and Paul Bettega for their feedback on this work. This work has been partially supported by the EU Project STAMP ICT-16-10 No. 731529 under the H2020 framework program, by the Wallenberg Autonomous Systems and Software Program, by the TrustFull project financed by the Swedish Foundation for Strategic Research and by the OSS-Orange-Inria project.

Appendix

Add method invocation

The following section details the transformation Add method invocation. Listing "Add method invocation" section describes the subset of the Java language targeted, and what follows describes the transformation's behavior.

```
<class> ::= (<modifier>)*
'class' <identifier> (ref)*
'{' (<class_member>)* '}'

<class_member> ::= <attribute> |
<other> | <method>

<method> ::= (<modifier>)* (return_type)
<identifier> '('
(<type> <identifier>)* ')' '{'
(<statement>)* '}'

<statement> ::= <variable_declaration> |
<block> | <other>

<block> ::= (<block_header>) '{'
(<statement>)* '}'
```

$P = \{\text{Packages}\}$,
$C(p) = \{\text{Classes of } p \| p \in P\}$,
$M(c) = \{\text{methods in class } c\}$,
$S(m) = \{\text{statements in method } m\text{'s body}\}$,

$LV(s) = \{\text{Local variables up to } s\}$,
$Pa(m) = \{\text{Parameter of method } m\}$,
$A(c) = \{\text{Attributes of class } c\}$,
$As(c) = \{a \in A(c) \| static(a)\}$,
Let $p \in P, c \in C(p), m \in M(c), s \in S(m)$
$V(c, m, s) = LV(s) \cup Pa(m) \cup A(c) \cup \{this\}$, a set of accessible variable from s
$Vs(c, m, s) = LV(s) \cup Pa(m) \cup As(c)$, a set of accessible variable from s
$\mathcal{M}(p, c, m, s) = \{m' \in c\} \cup \{m' \in c' \| \forall c' \in p \wedge \neg private(m')\} \cup \{m' \in c' \| \forall p' \in P, \forall c' \in p' \wedge public(m')\}$
$\mathcal{M}_a(p, c, m, s) = \{m' \| static(m') \vee ClassOf(m') \in TypeOf(LV(c, m, s))\}$

Let $m' \in \mathcal{M}_a$

$$\text{Class}\frac{c \text{ follows } ...m...}{c \rightarrow ...Well; m...}$$

$$\text{Well}_{static \wedge \neg void}\frac{\text{static}(m) \ \wedge \ \text{TypeOf}(m') \neq void}{Well \rightarrow \text{'public static' TypeOf}(m') \text{ wellID}}$$

$$\text{Well}_{\neg static \wedge \neg void}\frac{\neg \text{ static}(m) \ \wedge \ \text{TypeOf}(m') \neq void}{Well \rightarrow \text{'public' TypeOf}(m') \text{ wellID}}$$

$$\text{Well}_{void}\frac{\text{TypeOf}(m') = void}{Well \rightarrow SKIP}$$

$$\text{Method}\frac{m \text{ follows } ...s...}{m \rightarrow ...\text{'try } \{'Well\ Ta\ Call;'\} \text{ catch (Exception ' eId ') } \{\}'s...}$$

$$\text{We}\frac{\text{TypeOf}(m') \neq void}{We \rightarrow wellID =}$$

$$\text{We}_{void}\frac{\text{TypeOf}(m') = void}{We \rightarrow SKIP}$$

$$\text{Target}_{static}\frac{\text{static}(m')}{Ta \rightarrow SKIP}$$

$$\text{Target}\frac{\neg \text{ static}(m')}{Ta \rightarrow targetID.}$$

$$\text{Call}\frac{}{Call \rightarrow QN(m')(params)}$$

SwapSubtype

The following section details the behavior of the SwapSubtype transformation, the subset of Java targeted "SwapSubtype" section, and how it modified it.

```
<affectation> ::= <ls> '=' <rs>
<ls> ::=  (<interface> ('<' <type> '>')?)? <identifier>
<rs> ::= 'new' <concrete_class_constructor> ('<' <type> '>')? '('
    <param_list> ')'
<param_list> ::= <> | <param> | <param> ',' <param_list>
\label{lst:col-assign}
```

$I = \{\text{Interfaces}\}$
$T = \{\text{Types}\}$
$C(t) = \{\text{Constructor of } t\}$
$T(i) = \{t \in T \| t \text{ implements } i\}$

Let $i \in I, t_1, t_2 \in T(i)^2$ such as $t_1 \neq t_2$

$$\text{Affectation}\frac{\text{Aff}(I, id, t_1, params) \wedge \exists c \in C(t_2)}{\text{Aff}(I, id, t_1, params) \rightarrow Iid =' new' t_2'('params')'}$$

The following sections list the different interfaces targeted by our implementation of the transformation, and for each interface the different classes implementing these interfaces used interchangeably.

java.util.SortedSet

- java.util.concurrent.ConcurrentSkipListSet
- java.util.TreeSet

java.util.concurrent.BlockingDeque

- java.util.concurrent.LinkedBlockingDeque

java.util.Collection

- java.util.concurrent.LinkedTransferQueue
- java.util.concurrent.SynchronousQueue
- java.util.PriorityQueue
- java.util.concurrent.CopyOnWriteArraySet
- java.util.concurrent.LinkedBlockingQueue
- java.util.TreeSet
- java.util.concurrent.ConcurrentLinkedDeque
- java.util.Stack
- java.util.concurrent.PriorityBlockingQueue
- java.util.ArrayList
- java.util.HashSet
- java.util.concurrent.ArrayBlockingQueue
- java.util.Vector

- java.util.concurrent.ConcurrentSkipListSet
- java.util.concurrent.LinkedBlockingDeque
- java.util.concurrent.DelayQueue
- java.util.ArrayDeque
- java.util.LinkedList
- java.util.LinkedHashSet
- java.util.concurrent.ConcurrentLinkedQueue
- java.util.concurrent.CopyOnWriteArrayList

java.util.concurrent.ConcurrentNavigableMap

- java.util.concurrent.ConcurrentSkipListMap

java.util.Set

- java.util.HashSet
- gnu.trove.set.hash.THashSet
- java.util.concurrent.ConcurrentSkipListSet
- org.apache.commons.collections4.set.ListOrderedSet
- java.util.concurrent.CopyOnWriteArraySet
- java.util.TreeSet
- java.util.LinkedHashSet
- gnu.trove.set.hash.TCustomHashSet

java.util.concurrent.BlockingQueue

- java.util.concurrent.ArrayBlockingQueue
- java.util.concurrent.LinkedTransferQueue
- java.util.concurrent.SynchronousQueue
- java.util.concurrent.LinkedBlockingDeque
- java.util.concurrent.DelayQueue
- java.util.concurrent.LinkedBlockingQueue
- java.util.concurrent.PriorityBlockingQueue

java.util.NavigableSet

- java.util.concurrent.ConcurrentSkipListSet
- java.util.TreeSet

java.util.Deque

- java.util.concurrent.LinkedBlockingDeque
- java.util.ArrayDeque
- java.util.LinkedList
- java.util.concurrent.ConcurrentLinkedDeque

java.util.concurrent.TransferQueue

- java.util.concurrent.LinkedTransferQueue

java.util.NavigableMap

- java.util.concurrent.ConcurrentSkipListMap
- java.util.TreeMap

java.util.concurrent.ConcurrentMap

- java.util.concurrent.ConcurrentSkipListMap
- java.util.concurrent.ConcurrentHashMap

java.util.List

- org.apache.commons.collections4.list.TreeList
- java.util.Vector
- org.apache.commons.collections4.list.NodeCachingLinkedList
- org.apache.commons.collections4.list.CursorableLinkedList
- java.util.LinkedList
- org.apache.commons.collections4.list.GrowthList
- java.util.Stack
- java.util.ArrayList
- java.util.concurrent.CopyOnWriteArrayList
- org.apache.commons.collections4.ArrayStack

java.util.Map

- org.apache.commons.collections4.map.SingletonMap
- org.apache.commons.collections4.map.Flat3Map
- org.apache.commons.collections4.map.LinkedMap
- java.util.concurrent.ConcurrentHashMap
- org.apache.commons.collections4.map.LRUMap
- org.apache.commons.collections4.map.ListOrderedMap
- java.util.HashMap
- org.apache.commons.collections4.map.HashedMap
- org.apache.commons.collections4.map.ReferenceMap
- org.apache.commons.collections4.map.CaseInsensitiveMap
- gnu.trove.map.hash.TCustomHashMap
- java.util.LinkedHashMap
- org.apache.commons.collections4.map.PassiveExpiringMap
- java.util.concurrent.ConcurrentSkipListMap
- org.apache.commons.collections4.map.StaticBucketMap
- java.util.TreeMap
- gnu.trove.map.hash.THashMap

- java.util.Hashtable
- java.util.WeakHashMap
- org.apache.commons.collections4.map.ReferenceIdentityMap

java.util.Iterable

- java.util.concurrent.LinkedTransferQueue
- java.util.concurrent.SynchronousQueue
- java.util.PriorityQueue
- java.util.concurrent.CopyOnWriteArraySet
- java.util.concurrent.LinkedBlockingQueue
- java.util.TreeSet
- java.util.concurrent.ConcurrentLinkedDeque
- java.util.Stack
- java.util.concurrent.PriorityBlockingQueue
- java.util.ArrayList
- java.util.HashSet
- java.util.concurrent.ArrayBlockingQueue
- java.util.Vector
- java.util.concurrent.ConcurrentSkipListSet
- java.util.concurrent.LinkedBlockingDeque
- java.util.concurrent.DelayQueue
- java.util.ArrayDeque
- java.util.LinkedList
- java.util.LinkedHashSet
- java.util.concurrent.ConcurrentLinkedQueue
- java.util.concurrent.CopyOnWriteArrayList

java.util.Queue

- java.util.concurrent.LinkedTransferQueue
- java.util.concurrent.SynchronousQueue
- java.util.PriorityQueue
- java.util.concurrent.LinkedBlockingQueue
- java.util.concurrent.ConcurrentLinkedDeque
- java.util.concurrent.PriorityBlockingQueue
- java.util.concurrent.ArrayBlockingQueue
- org.apache.commons.collections4.queue.CircularFifoQueue
- java.util.concurrent.LinkedBlockingDeque
- java.util.concurrent.DelayQueue
- java.util.ArrayDeque
- java.util.LinkedList
- java.util.concurrent.ConcurrentLinkedQueue

java.util.SortedMap

- java.util.concurrent.ConcurrentSkipListMap
- java.util.TreeMap

Loopflip

```
<loop> ::= 'for(' <initialization> ';'
<condition> ';' <update> ')' '{'
(<statement>)* '}'

<initialization> ::= <identifier>
'=' <expression>

<condition> ::= <identifier>
<binary_operator> <expression>

<binary_operator> ::= '<' |
'<=' | '>' | '>='

<update> ::= <identifier> '='
<identifier> <operator> <expression>

<operator> ::= '+' | '-'
```

We extends update statements such as `i++` into `i = i + 1` and `i -= 2` into `i = i - 2`

$$\text{comp} \in \{<, >, \geq, \leq\}, \text{op} \in \{+, -\}$$

$$\overline{a}, \forall a \in \{<, >, \geq, \leq\} \cup \{+, -\} \begin{cases} >, \geq \mapsto \leq \\ <, \leq \mapsto \geq \\ + \mapsto - \\ - \mapsto + \end{cases}$$

$$\text{For}_L \frac{\text{comp} \in \{\geq, \leq\} \wedge |i_{end} - i_0| \equiv 0 \pmod p}{(\text{For}_L(i = i_0 i \text{ comp } i_{end} i = i \text{ op } p) \rightarrow (\text{For}_L(i = i_{end} i \ \overline{\text{comp}} \ i_0 i = i \ \overline{\text{op}} \ p)}$$

$$\text{For}_L \frac{\text{comp} \notin \{\geq, \leq\} \vee \neg(|i_{end} - i_0| \equiv 0 \pmod p)}{(\text{For}_L(i = i_0 i \text{ comp } i_{end} i = i \text{ op } p) \rightarrow (\text{For}_L(i = i_{end} \ \overline{\text{op}} \ (|i_{end} - i_0| \pmod p)) i \ \overline{\text{comp}} \ i_0 i = i \ \overline{\text{op}} \ p)}$$

References

1. S. Allier, O. Barais, B. Baudry, J. Bourcier, E. Daubert, F. Fleurey, M. Monperrus, H. Song, M. Tricoire, Multi-tier diversification in web-based software applications. IEEE Softw. **32**, 83–90 (2015)
2. E.T. Barr, M. Harman, Y. Jia, A. Marginean, J. Petke, Automated software transplantation, in *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (ACM, 2015), pp. 257–269
3. M. Basios, L. Li, F. Wu, L. Kanthan, E.T. Barr, Darwinian data structure selection, in *Proceedings of ESEC/FSE* (2018)

4. B. Baudry, S. Allier, M. Monperrus, Tailored source code transformations to synthesize computationally diverse program variants, in *Proceedings of ISSTA* (2014), pp. 149–159
5. B. Baudry, M. Monperrus, The multiple facets of software diversity: recent developments in year 2000 and beyond. ACM Comput. Surv. **48**(1), 16:1–16:26 (2015)
6. R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab et al., Grid'5000: a large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. **20**(4), 481–494 (2006)
7. A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, Cross-checking oracles from intrinsic software redundancy, in *Proceedings of ICSE, ICSE 2014* (2014), pp. 931–942
8. A. Carzaniga, A. Mattavelli, M. Pezzè, Measuring software redundancy, in *Proceedings of ICSE*, vol. 1 (2015), pp. 156–166
9. S. Chiba, Load-time structural reflection in java, in *Proceedings of ECOOP*, (Springer, 2000), pp. 313–336
10. B. Cody-Kenny, M. O'Neill, S. Barrett, Performance localisation, in *Proceedings of GI* (2018)
11. F.B. Cohen, Operating system protection through program evolution. Comput. Secur. **12**(6), 565–584 (1993)
12. B. Danglot, P. Preux, B. Baudry, M. Monperrus, Correctness attraction: a study of stability of software behavior under runtime perturbation. Empir. Softw. Eng. **23**(4), 2086–2119 (2017)
13. E.F. de Souza, C.L. Goues, C.G. Camilo-Junior, A novel fitness function for automated program repair based on source code checkpoints, in *Proceedings of GECCO* (IEEE, 2018), pp. 1443–1450
14. P.S. Devan, R. Kamat, A review-loop dependence analysis for parallelizing compiler. Int. J. Comput. Sci. Inf. Technol. **5**(3), 4038–4046 (2014)
15. R. Feldt, Generating diverse software versions with genetic programming: an experimental study. IEE Proc. Softw. **145**(6), 228–236 (1998)
16. M. Gabel, Z. Su, A study of the uniqueness of source code, in *Proceedings of FSE* (ACM, 2010), pp. 147–156
17. S.O. Haraldsson, J.R. Woodward, A.E.I. Brownlee, A.V. Smith, V. Gudnason, Genetic improvement of runtime and its fitness landscape in a bioinformatics application, in *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO'17, New York, NY, USA* (ACM, 2017), pp. 1521–1528
18. M. Harman, B.F. Jones, Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001)
19. N. Harrand, B. Baudry, Software diversification as an obfuscation technique, in *International Workshop on Obfuscation: Science, Technology, and Theory* (2017), pp. 31–34
20. D. Kim, Y. Kwon, W.N. Sumner, X. Zhang, D. Xu, Dual execution for on the fly fine grained execution comparison. SIGPLAN Not. **50**(4), 325–338 (2015)
21. W.B. Langdon, J. Petke, Software is not fragile, in *First Complex Systems Digital Campus World E-Conference 2015* (Springer, 2017), pp. 203–211
22. W.B. Langdon, N. Veerapen, G. Ochoa, Visualising the search landscape of the triangle program, in *Genetic Programming*, ed. by J. McDermott, M. Castelli, L. Sekanina, E. Haasdijk, P. García-Sánchez (Springer, Cham, 2017), pp. 96–113
23. J. López, N. Kushik, N. Yevtushenko, Source code optimization using equivalent mutants. CoRR, abs/1803.09571 (2018)
24. I. Manotas, L. Pollock, J. Clause, Seeds: a software engineer's energy-optimization decision support framework, in *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, New York, NY, USA* (ACM, 2014), pp. 503–514
25. S. Mittal, A survey of techniques for approximate computing. ACM Comput. Surv. **48**(4), 62 (2016)
26. H. Okhravi, T. Hobson, D. Bigelow, W. Streilein, Finding focus in the blur of moving-target techniques. IEEE Secur. Priv. Mag. **12**(2), 16–26 (2014)
27. R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, L. Seinturier, Spoon: a library for implementing analyses and transformations of java source code. Softw. Pract. Exp. **46**, 1155–1179 (2015)
28. J. Petke, S. Haraldsson, M. Harman, D. White, J. Woodward et al., Genetic improvement of software: a comprehensive survey. IEEE Trans. Evolut. Comput. **22**(3), 415–432 (2017)
29. J. Renzullo, W. Weimer, M. Moses, S. Forrest, Neutrality and epistasis in program space, in *Proceedings of GI* (ACM, 2018)
30. M. Rinard, Obtaining and reasoning about good enough software, in *Proceedings of DAC* (ACM, 2012), pp. 930–935

31. M. Rodriguez-Cancio, B. Combemale, B. Baudry, Automatic microbenchmark generation to prevent dead code elimination and constant folding, in *Proceedings of ASE* (Singapore, Singapore, 2016)
32. A. Sălcianu, M. Rinard, Purity and side effect analysis for Java programs, in *Verification, Model Checking, and Abstract Interpretation*, ed. by R. Cousot (Springer, Berlin, 2005), pp. 199–215
33. E. Schulte, J. Dorn, S. Harding, S. Forrest, W. Weimer, Post-compiler software optimization for reducing energy, in *ACM SIGARCH Computer Architecture News*, vol. 42 (ACM, 2014), pp. 639–652
34. E. Schulte, Z.P. Fry, E. Fast, W. Weimer, S. Forrest, Software mutational robustness. Genet. Program. Evolvable Mach. **15**(3), 281–312 (2014)
35. O. Shacham, M. Vechev, E. Yahav, Chameleon: adaptive selection of collections. SIGPLAN Not. **44**(6), 408–418 (2009)
36. F.J. Shull, J.C. Carver, S. Vegas, N. Juristo, The role of replications in Empirical Software Engineering. Empir. Softw. Eng. **13**(2), 211–218 (2008)
37. M. Suzuki, A.C. de Paula, E. Guerra, C.V. Lopes, O.A.L. Lemos, An exploratory study of functional redundancy in code repositories, in *Proceedings of SCAM* (IEEE, 2017), pp. 31–40
38. S. Wang, P. Wang, D. Wu, Composite software diversification, in *Proceedings of ICSME* (IEEE, 2017), pp. 284–294
39. X. Yao, M. Harman, Y. Jia, A study of equivalent and stubborn mutation operators using human analysis of equivalence, in *Proceedings of ICSE* (2014), pp. 919–930
40. Y. Yuan, W. Banzhaf, Arja: automated repair of Java programs via multi-objective genetic programming. IEEE Trans. Softw. Eng. (2018). https://doi.org/10.1109/TSE.2018.2874648

Affiliations

Nicolas Harrand[1] · Simon Allier[2] · Marcelino Rodriguez-Cancio[3] · Martin Monperrus[1] · Benoit Baudry[1]

Simon Allier
simon.allier@intradef.gouv.fr

Marcelino Rodriguez-Cancio
marcelino.rguez.cancio@gmail.com

Martin Monperrus
martin.monperrus@csc.kth.se

[1] KTH, Stockholm, Sweden

[2] DGA, Val-de-Reuil, France

[3] Vanderbildt University, Nashville, USA