On Understanding Contextual Changes of Failures

Francisco Ribeiro
HASLab/INESC TEC
University of Minho
Braga, Portugal
francisco.j.ribeiro@inesctec.pt

Rui Abreu
INESC-ID & FEUP
University of Porto
Porto, Portugal
rui@computer.org

João Saraiva
HASLab/INESC TEC
University of Minho
Braga, Portugal
saraiva@di.uminho.pt

***Abstract*—Recent studies show that many real-world software faults are due to slight modifications (mutations) to the program. Thus, analyzing transformations made by a developer and associating them with well-known mutation operators can help pinpoint and repair the root cause of failures. This paper proposes a mutation operator inference technique: given the original program and one of its subsequent forms, it infers which mutation operators would transform the original and produce such a version. Moreover, we implemented this technique as a tool called *Morpheus*, which analyzes faulty Java programs. We have also validated both the technique and tool by analyzing a repository with 1753 modifications for 20 different programs, successfully inferring mutation operators 78% of times. Furthermore, we also show that several program versions result from not just a single mutation operator but multiple ones. In the end, we resort to real-world case studies to demonstrate the advantages of this approach regarding program repair.**

***Index Terms*—program comprehension, program mutation, program repair**

I. Introduction

Software development methodologies have quickly evolved in recent years. Teams of engineers develop complex and large software applications in a collaborative environment, using version control systems, testing frameworks and continuous integration mechanisms to improve productivity. As is increasingly common in such a development environment, a new software feature or maintenance update needs to be *pushed* (to the version control system) to be available to the development team. After such a push, the continuous integration mechanism performs a set of tests that the new software ought to pass. When all the tests pass, a new software build is produced. However, often the software does not pass such tests because of a bug that was unintentionally introduced. Developers typically look at the source code to identify changes between versions that may have introduced the bug. This analysis is frequently based on *diff* reports and misses the context of the fault. Let us consider an example taken from the *Bugswarm* [1] repository containing real-world bugs and their fixes:

```
132c133
<     ... property.toUpperCase());
---
>     ... property.toUpperCase(Locale.ENGLISH));
```

Simply knowing that line 132 got modified does not reflect the modifications' *semantics* (context), making it difficult to understand and fix the problem without concrete clues. If we analyze further, we see that this modification essentially adds a parameter in an already existing method call. More precisely, `property.toUpperCase()` now gets an additional parameter, represented by the call `property.toUpperCase(`**`Locale.ENGLISH`**`)`. An *argument number change* [2, 3, 4] is an example of a mutation that changes the number of input parameters in method invocations, given that there is a definition for the same method name which accepts the new arguments. Although this is a simple motivating example, recent studies show that many real-world software faults are coupled to mutation operators [5, 6, 7, 8].

This paper presents a mutation operator inference technique that, given the original program and one of its subsequent versions, infers the mutation operators capable of producing such an alternative. This is achieved by interpreting the changes made to the abstract syntax tree (AST). Although the different program versions are obtained by manually modifying the source code, as it happens in a real-world software project, throughout this paper, we sometimes refer to them as *mutants*, even though there was no mutation testing tool involved.

The contributions of this paper are: 1) a technique that allows for the detection/inference of mutation operators based on AST transformations; 2) a tool implementing this technique; 3) a dataset produced by validating our technique and tool over an existing repository of 1753 manually modified programs with information about the detected mutation operators; 4) a repair strategy that reverts the detected mutation operators and a tool implementing it; 5) a case study investigation with real-world bugs from the *Bugswarm* and *Defects4J* repositories showing how this work can benefit program repair.

Being able to infer the mutation operators is a first step to incorporate automated program repair in a continuous integration system, where a faulty program is fixed by considering the *contextual modifications* that led to the introduction of the bug. By analyzing a considerable number of programs, we can verify the most common mutations. As far as we know, there is no ranking that lists each mutation operator's frequency in a real-world scenario. That is, the number of times a manually created version of a program translates to the application of commonly known mutation operators.

While mutation testing focuses on injecting faults as small modifications, our work aims to analyze the prevalence of those exact patterns of modifications in source code modified by humans.

II. Mutation Analysis

Mutation operators are one of the pillars of *mutation testing* [9], a technique that introduces faults in source code to assess the quality of tests. This quality is measured by evaluating the test suite's ability to detect the mutated programs. That is, tests that cover the mutated code should fail. Mutation testing relies on the quality of mutation operators and their aim is to mimic programming errors, such as using the wrong value for a constant, applying an incorrect binary operator or referring to a wrong variable's name. The alternative programs produced by these operators (called *mutants*) are semantically correct, i.e., the program is valid. Research concerning mutation operators has been widely conducted. As a result, many operators representing precise transformations have been defined. Although the study of mutation operators started by targeting general programming aspects [10, 11], the definition of such operators can be more specific, with some works describing mutations specialized for object-oriented settings [2, 12]. The mutation operators considered in this work have been taken from previous literature and incorporated into existing mutation testing tools [9, 12, 13].

TABLE I: Mutation operators: possible inferences

Mutation Operator	**Example**
`ConstantReplacement`	int i = **0** → int i = **1**
`RelationalOperatorReplacement`	x **<=** 2 → x **<** 2
`VarToVarReplacement`	next = **var1** → next = **var2**
`StatementDeletion`	~~int n = 1;~~
`ArithmeticOperatorInsertion`	int a = b; → int a = b **+ 1**;
`NonVoidMethodDeletion`	String s ~~= getName();~~
`VarToConsReplacement`	int i = **j**; → int i = **0**;
`ReturnValue`	return **2**; → return **3**;
`UnaryOperatorInsertion`	setX(x); → setX(x**++**)
`ConditionalOperatorReplacement`	x<=2 **&&** y<4 → x<=2 || y<4
`VoidMethodDeletion`	~~print("str");~~
`ConditionalOperatorDeletion`	x<=2 **&&** y<4 → x<=2
`ArithmeticOperatorReplacement`	float x = a * b → float x = a **/** b;
`MemberVariableAssignmentDeletion`	private int x ~~= 3~~;
`AccessorModifierChange`	**public** void... → **private** void...
`UnaryOperatorReplacement`	**i++** → **i- -**
`RemoveConditional`	if(**x < 2**) → if(**true**)
`ArithmeticOperatorDeletion`	float x = a ***** b → float x = a;
`ConsToVarReplacement`	int x = **2**; → int x = **a**;
`ConditionalOperatorInsertion`	x<=2 → x<=2 **&&** y<4
`UnaryOperatorDeletion`	setX(x**++**) → setX(x);
`ConstructorCallReplacementNull`	String s = ~~new String()~~ null;
`StaticModifierDeletion`	public **static** int... → public int...
`AccessorMethodChange`	point.**getX**(); → point.**getY**();
`BitshiftOperatorReplacement`	1 **«** 30 → 1 **»** 30
`ReferenceReplacementContent`	someObj → someObj**.clone()**
`StaticModifierInsertion`	public int... → public **static** int...
`TrueReturn`	return **x<=2**; → return **true**;
`FalseReturn`	return **x<=2**; → return **false**;
`ArgumentTypeChange`	method(int x){ → method(long x){
`ArgumentNumberChange`	new Person(); → new Person(**"joe"**);
`BitshiftOperatorDeletion`	1 **«** 30 → 1
`BitwiseOperatorReplacement`	int x = a | b; → int x = a **&** b;
`Negation`	int x = num; → int x = **-**num;

Test coverage is not enough to guarantee a test suite's quality, as it is purely a quantitative measure of the amount of source code we exercise. Having a way of automatically generating alternatives to our programs is helpful, as now we can qualitatively measure if our test suite is robust enough to react to the presence of bugs. Moreover, interpreting the source code and manually creating mutants of the most pertinent parts of the program's logic would be very inefficient. However, there is still the question of whether artificial faults, i.e., mutants, are a suitable replacement for real faults. Several studies [5, 6, 7, 8] have analyzed the connection between real faults, i.e., *bugs* accidentally introduced while developing a real-world application, and mutants. In general, results obtained by using real errors are also obtained by using mutants. In particular, one of these studies [5] even shows that real faults are coupled with commonly used mutation operators by mutation testing frameworks. This means that real faults can be translated as the application of well-known mutation operators. Thus, we argue that if we better understand how software was modified in terms of the application of mutation operators, we can efficiently design a repair strategy that fixes the program. The changes introduced by these operators represent small modifications to a program's logic and can be interpreted as changes to the program's *structure*. Therefore, to accurately detect these modifications, we focus on the structural representation of programs. The AST represents a program's source code in which emphasis is given to structure and contents. The nodes composing such trees represent constructs used in the corresponding program, e.g., *if's*, *while's*, *expressions*, etc. Therefore, when obtaining the set of modifications by comparing programs based on their AST's, we can better understand how its structure changed. These modifications are additions, deletions, updates or movements of nodes in the AST. Because these nodes have information related to the source code's context, we can see how a program changed *semantically*.

In terms of AST differencing, Figure 1 illustrates the introductory example[1]. As we can see, the area pointed at shows where an *insert operation* was performed. By capturing this change and verifying the context in which it occurs, we can realize its actual meaning. In this case, the change is applied to an invocation and the number of its arguments gets modified. As such, we can infer it translates to applying an `ArgumentNumberChange` operator. Creating a technique that embodies this reasoning allows us to derive the *semantics* of source code modifications automatically. As a result, developers can get support in reasoning why certain changes lead to errors.

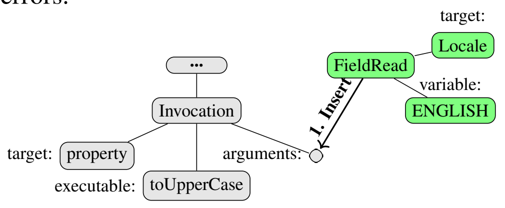

Fig. 1: Argument number change: introductory example

III. Inferring mutations

To infer the mutation operators responsible for a new program version, we focus on the AST representations of both the original and the mutated versions. This approach allows us to circumvent a significant limitation concerning *textual diffs*: we lose the code's structural notion, i.e., what each piece represents and how it connects to the remaining parts of the

[1]Colors represent the action types applied to the AST: Green - Insert; Red - Delete; Yellow - Move; Orange - Update

source code. Furthermore, detecting changes at a textual level regarding the evolution of a file mainly considers two possible representations: insertions and deletions. Detecting changes made to structured data is a topic that has been subject to a considerable amount of research [14, 15, 16]. This set of changes, called an *edit script*, is computed at a node level and considers more types of transformations: insertions, deletions, updates and moves. In short, given two trees, T_1 and T_2, we are interested in obtaining the edit script, which, when applied to T_1, produces T_2. Consider the following original code and corresponding mutant:

```
return h & (length - 1); //original
return h & (length);     //mutant
```

In mutation testing, we could obtain the previous mutant by applying the `ArithmeticOperatorDeletion` operator. Detecting such modifications based on the textual representation of this part of the source code would be cumbersome. We only know which lines changed and we have no information about which part of the line was modified. Moreover, we would have to parse a partial program, which is a task subject to ambiguities [17]. There is no sensible way of parsing incomplete code without rapidly falling into errors. At some point, the degree of incompleteness will easily cause the failure of whatever workaround strategies the parser is using.

In turn, if we provide the two complete programs to an AST *diff* tool[2], we obtain a set of transformations, which Figure 2 illustrates.

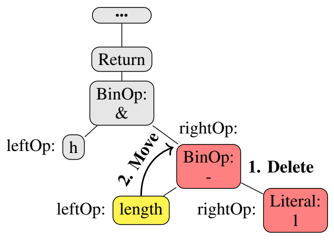

Fig. 2: Arithmetic operator deletion

Because an AST conveys the program's structure, instead of being directed to a line in the source code, the differencing algorithm analyzes the tree nodes and can detect changes at the component granularity. Also, because tokens are split into their corresponding nodes, carrying their meaning in the source code, we can isolate changes at a more elementary level. Here, a total of two operations are involved:

- a **delete operation** of the node corresponding to the subtraction operator, which in turn implies the deletion of one of its operands, in this case, the literal 1;
- a **move operation** of the leftover operand, which now takes the spot of the deleted one.

However, checking only for the types of operations, in this case, *delete* and *move*, is not sufficient. The simple occurrence of such node modifications can have different meanings, as other mutation operators also manifest themselves through these kinds of AST transformations. In order to correctly pinpoint this case as the application of an *arithmetic operator deletion*, we need:

- to check if the **deleted node** is a binary operator and if it is an arithmetic operator (in this case: "-");
- to check if the **moved node** is one of the operands of the deleted node (in this case: "length").

Of course, when we only have the difference between ASTs, we do not know which mutation operators are present. As such, we need to check against all possible operators until we exhaust all options. Similar to what we previously described regarding the *arithmetic operator deletion*, each mutation operator is associated with a set of rules that must be observed to establish its presence accurately.

A. Multiple mutations

A file may be modified in a way that reflects several mutation operator applications at once. Jia and Harman [18] define this concept as *higher-order mutants*, referring to them as the injection of two or more simple faults, which they call *first-order mutants*. However, the AST *diff* does not differentiate groups of transformations and it produces a set with all transformations. We subdivide this set into subsets so that we can better isolate the occurrences of operators.

1) Overlapping mutations: Let us consider the textual diff:

```
173c173
<     if (inflection.match(word)) {
---
>     if (true) {
```

In this example, we have at least two mutation operators:

- `NonVoidMethodDeletion`: removes a call to a non-void method; in this case, the call to `match()` was deleted;
- `RemoveConditional`: removes conditional expressions with either *true* or *false*; in this case, the expression `inflection.match(word)` got replaced by `true`.

The two operators overlap and, thus, we need to take special care when analyzing the following transformations to the AST:

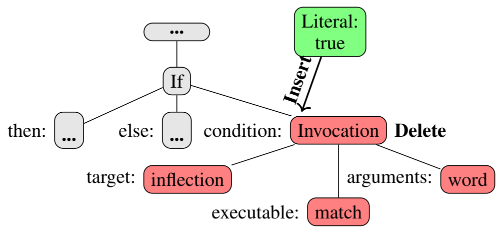

Fig. 3: Two overlapping mutation operators

As we can see, there is:

- a **delete operation** of the method invocation node inside the `if` condition, which deletes its children;
- an **insert operation** of the literal value `true` in the place where the previous deletion occurred.

As mentioned before, two operators overlap:

- `NonVoidMethodDeletion` is identified by the delete operation, for which we check if the deleted node corresponds to an *Invocation* node and if the return type is not void (*boolean* in this case);
- `RemoveConditional` is identified by both operations - delete and insert. For this, we need to check if the deleted

[2]https://github.com/SpoonLabs/gumtree-spoon-ast-diff

node's parent is of *If* type, if the inserted one is a *Literal* node and its value is a *boolean* (here, *true*) and if the inserted node is placed in the same spot as the previously deleted node, i.e., if both nodes' parent (*If*) is the same.

2) Independent mutations: Sometimes, the different places in a program where mutations are applied are independent of one another. Similarly, analyzing fragments of the complete set of AST modifications helps in detecting all the various operators.

```
191c191
<   int oldCapacity = oldTable.length;
---
>   int oldCapacity = oldTable.length-1;
260c260
<   next = n;
---
>   next = k;
303c303
<   if (x == y) {
---
>   if (x != y) {
```

In this example, three mutation operators were applied in entirely different places of the program:

- `ArithmeticOperatorInsertion`: an arithmetic operator (`+, -, *, /, %`) is inserted, performing an operation between an existing variable/constant in the code and an inserted operand; here, the expression `oldTable.length` became `oldTable.length - 1` (Figure 4);
- `VarToVarReplacement`: a variable's name is replaced by another one; here, variable `k` replaces `n` (Figure 5);
- `RelationalOperatorReplacement`: a relational operator (`>, >=, <, <=, ==, !=`) is replaced by a different one; here, $==$ was replaced by $!=$ (Figure 6).

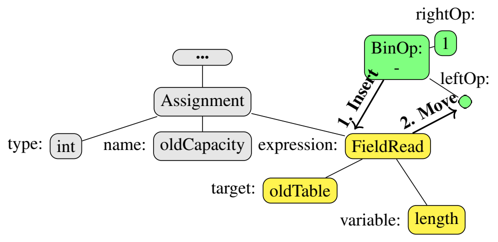

Fig. 4: Arithmetic operator insertion

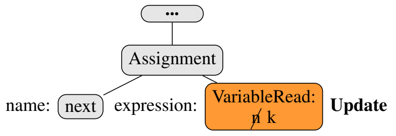

Fig. 5: Variable replacement

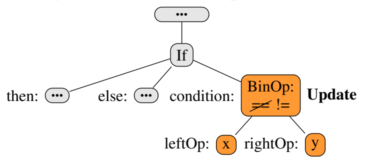

Fig. 6: Relational operator replacement

All three previous images illustrate the modified parts of the complete AST. For each mutation operator, we need to check for different properties/rules:

- `ArithmeticOperatorInsertion`: check if the inserted node is a *BinaryOperator* and if its type is arithmetic. We also need to verify if the inserted node is taking the spot of the moved node. Additionally, it is necessary to confirm that neither of the operands is a *String*, as the binary operator $+$ is also used for string concatenation;
- `VarToVarReplacement`: check if the update operation is being performed on a *VariableRead* node;
- `RelationalOperatorReplacement`: check if the update operation is performed on a *BinaryOperator* node and if it is of *relational* kind.

IV. Inference Technique and Tool

This section describes the technique's algorithm and its implementation in the *Morpheus* tool.

A. Algorithm

The algorithm for inferring mutation operators from AST transformations can be divided into two phases (Algorithm 1): 1) **partitioning —** subdivides the set of AST operations so that each resulting subpart can be analyzed separately, isolating it from the "noise" of all other transformations; 2) **matching —** the subsets from the previous phase are matched against the mutation operator patterns. An analogy can be made regarding *type inference* systems, where expressions are associated with their correct types via automatic inference. Here, a set of transformations are "expressions" for which we are trying to associate with mutation operators ("types"). As we do not have any specific information regarding the nature of the mutation (which would correspond to type annotations in languages using *manifest typing*), we have to detect specific properties (*type rules*) in the provided transformations to conclude what mutation operator it represents.

Algorithm 1: Algorithm for inferring mutation operators

```
Data: An AST Diff between trees $T_1$ and $T_2$ containing the list of
      modifications $M$ which, if applied to $T_1$, will produce $T_2$; a
      list of mutation operators $MO$ to search for; an empty list of
      inferred mutation operators $IMO$
Result: The list of inferred mutation operators $IMO$
1  subLists ← [];
2  for i ← 1,size($M$) do
3      add([$M_i$], subLists); // appends list to subLists
4      for j ← i+1,size($M$) do
5          add([$M_i$, $M_j$], subLists);
6          for k ← j+1,size($M$) do
7              add([$M_i$, $M_j$, $M_k$], subLists);
8          end
9      end
10 end
11 foreach list ∈ subLists do
12     foreach $mo \in MO$ do
13         inferred ← matches(mo, list);
14         if inferred != null then
15             add(inferred, IMO);
16         end
17     end
18 end
```

To better illustrate the first part of the algorithm, let us consider a list of the form $[\mathtt{Operation}_{node}]$:

$[\mathtt{Delete}_A,\ \mathtt{Insert}_B,\ \mathtt{Move}_C]$

This example indicates that we should apply one *delete*, one *insert* and one *move* operation to the original program tree to obtain the target program tree. Applying the previously mentioned sub-listing procedure would result in the sub-lists:

```
[ [Delete$_A$], [Delete$_A$, Insert$_B$],
 [Delete$_A$, Insert$_B$, Move$_C$], [Delete$_A$, Move$_C$],
 [Insert$_B$], [Insert$_B$, Move$_C$], [Move$_C$] ]
```

The first part of the algorithm is responsible for partitioning the complete list of AST modifications into sub-lists, each with a size between one and three (as these correspond to the minimum and the maximum number of changes that define a mutation operator, respectively). There are two reasons for this. First, because a program may have been modified in a way that reflects the application of more than one mutation operator, we need to identify multiple patterns in the complete list of alterations. As such, considering smaller portions of modifications enables us to find such patterns. Second, although the changes made to the AST are provided in *sequential* order, it does not mean that the ones characterizing the mutation operators are *contiguous*. The described approach allows us to consider *non-adjacent* sets of modifications in the search space.

The *matching* phase of the algorithm looks for predefined patterns in the list of changes to the AST. In such AST, different expressions of the language are represented in different subtrees. The subtrees are constructed according to the productions/constructors defining the respective expression. In languages that use *type inference* systems, the inference mechanism traverses these trees and takes a different approach according to the type of node it is visiting. In other words, the node's type influences what rules are checked for the system to try to infer the correct data type.

Let us consider a simple syntax for expressions. Here, expressions can be a number, a variable's name or a binary operator that allows for more sub-expressions.

```
Expression = Num n | VarName n | BinOp
BinOp = Expression Op Expression
Op = + | - | / | *
```

A type inference system should have a mechanism that will take one of the possible expressions and, depending on the type it has in the tree, carry on with the appropriate action to determine the data type.

```
infer(expr: Expression): Type
 switch(expr)
  Num n -> //check if n is Int, Float, ...
  VarName n -> //check scope for variable n
  BinOp e1 op e2 ->
   infer(e1);
   infer(e2);
```

As we can see by the pseudo-code of a hypothetical `infer` function, the `switch` statement will perform different checks for different node types. That is, each node type encompasses its own set of rules.

The algorithm we describe takes a similar approach, as we can see in line 13. Each mutation operator comprises the group of rules it will analyze to report if some series of transformations complies with them. Let us take the example of the `ConstantReplacement` mutation operator, which changes the value of a constant in the source code. One of the inference rules for this case is expressed like:

M = program modification with set of transformations **T**
$\mathbf{T} = \mathbf{T_1}, ..., \mathbf{T_k}$
$\mathbf{orig_k}$ = original node of $\mathbf{T_k}$
$\mathbf{new_k}$ = new node created by $\mathbf{T_k}$
$\mathbf{op_x}$ = operand of node **x**
$\mathbf{p_x}$ = parent node of element **x**

$$\frac{\Gamma \vdash size(T) = 1 \\ \Gamma \vdash T_1 : Update \qquad \Gamma \vdash orig_1 : Lit \qquad \Gamma \vdash new_1 : Lit}{\Gamma \vdash M : ConstantReplacement}$$

In this case, the first thing to do is to check the number of transformations to the tree. If T only has one transformation (T_1), this needs to be an *Update* operation and both the original node and the modified one have to be of type *Literal* (representing constants) to confirm the mutator. An example would be changing `methodCall(1)` to `methodCall(2)`.

This `ConstantReplacement` mutator can also be present through another pattern consisting of two transformations, T_1 and T_2. If so, we analyze the case where the constant value changed signals, e.g. from `methodCall(0)` to `methodCall(-1)`.

The inference rule for this situation can be expressed as:

$$\frac{\Gamma \vdash size(T) = 2 \qquad \Gamma \vdash T_1 : Delete \qquad \Gamma \vdash T_2 : Insert \\ \Gamma \vdash orig_1 : Lit \qquad \Gamma \vdash orig_2 : UnaryOp \\ \Gamma \vdash op_{orig_2} : Lit \qquad \Gamma \vdash p_{orig_1} = p_{T_2}}{\Gamma \vdash M : ConstantReplacement}$$

For this example, operations T_1 and T_2 need to be a *Delete* and an *Insert*, respectively. The *Delete* operation corresponds to removing the constant 0 from the code. As such, the node to which the deletion operation is applied, $orig_1$, needs to be of type *Literal*. Because the new value for the constant is -1, we have to consider this as the addition of two separate elements: a *unary operator* representing the *negative signal* and a *literal* representing the number 1. Following this line of thought, the insertion operation T_2 needs to be applied to a *UnaryOp* node, $orig_2$. Furthermore, we also need to check if the operand associated with the unary operator is a constant, that is, a *Literal* node. Note that, although two nodes are inserted, we only consider the insertion of the top-level one, the *UnaryOp*, as the *Literal* node corresponding to the value -1 is its child. Also, we need to check if this deletion and insertion occurred in the same spot in the tree, which means the parent node of the deleted one must be the same as the parent of the insertion operation, represented by the expression $p_{orig_1} = p_{T_2}$. Figure 7 illustrates these modifications.

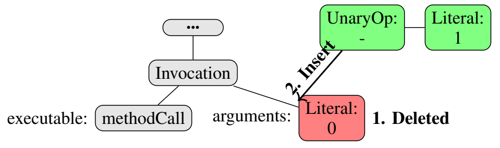

Fig. 7: Constant replacement - positive to negative value

B. Morpheus

We have implemented our technique as the *Morpheus* tool[3]. *Morpheus* analyzes *Java* programs and was developed using the *Kotlin* language. As shown in Figure 8, it consists of two components: The *Diff Calculation* gets as input the original and the mutated programs and produces the list of transformations representing the differences between the programs [16]. The second component - the *Inferrer* - implements Algorithm 1, with its two parts: *partitioning* and *matching*.

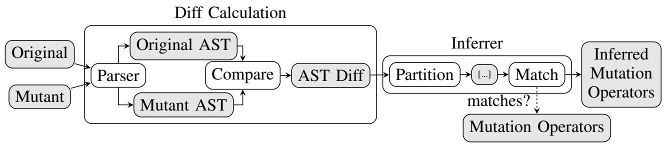

Fig. 8: *Morpheus* architecture

Morpheus is an extensible tool: it implements all operators in Table I and can easily be extended with new mutation operators due to its extensible architecture.

V. Dataset and Analysis

In this section, we present the structure of the produced dataset and analyze the results. *Morpheus* was used to analyze the *CodeDefenders* [19] repository[4] containing 20 original programs and 1753 mutants created by players. In *CodeDefenders*, attackers modify programs to introduce faults and defenders write unit tests that detect these errors. The 20 original classes are part of various real-world open-source projects. We identified 230 mutants that failed to compile and 27 for which the *AST diff* tool failed to produce an edit script, leading to 1496 valid programs.

A. Dataset Structure

After using our tool to analyze every valid mutant in the repository, we produced a dataset[5] containing information about each one. Therefore, each mutant has a corresponding record in the dataset with the following fields:

- **Mutant ID:** Mutant identifier based on the repository;
- **Nr. AST modifications:** Number of modifications applied to the original AST;
- **AST modifications:** List with the types of operations performed on the AST in order to obtain this mutant;
- **Mutation overviews:** List of the source code change for each inferred mutation operator;
- **Inferred mutation operators:** List containing the names of the inferred mutation operators (according to Table I);
- **Callables:** List containing the method/constructor names where each mutation was inferred;
- **Old start-end lines:** List containing the start and end lines in the *original* file where each mutation operator was detected (same index in inferred mutation operator list);
- **Old start-end columns:** Same as the previous field but for columns;
- **New start-end lines:** List containing the start and end lines in the *mutated* file where each mutation operator was detected;
- **Start-end columns:** Same as the previous field but for columns;
- **Relative old start-end lines:** List containing the start and end lines inside the callable's body in the *original* file where each mutation operator was detected;
- **Relative new start-end lines:** Same as the previous field but for the *mutated* file.

Recalling the example in Figure 2, the fields for the corresponding dataset record have the values shown in Table II.

TABLE II: Dataset record for one of the mutants

Mutant ID	`1001/15/00000001/ByteArrayHashMap`
Nr AST Modifications	`2`
AST Modifications	`[DeleteOperation, MoveOperation]`
Mutator Overviews	`[AOD(from=(length - 1),to=length)]`
Inferred Mutation Operators	`[ArithmeticOperatorDeletion]`
Callables	`[ByteArrayHashMap#indexFor(int,int)]`
Old Lines	`[324-324]`
Old Columns	`[16-27]`
New Lines	`[324-324]`
New Columns	`[14-21]`
Relative Old Lines	`[1-1]`
Relative New Lines	`[1-1]`

B. Dataset Analysis

Table III shows the number of mutants associated with each class in the repository. Furthermore, it also displays the number of mutants for which *Morpheus* could infer and classify as corresponding to one or more mutation operators.

TABLE III: Available mutants per valid program

Class	#Available Mutants	#Inferred Mutants	Effectiveness %
`ByteArrayHashMap`	126	108	86%
`ByteVector`	55	39	71%
`ChunkedLongArray`	95	68	72%
`FontInfo`	30	22	73%
`FTPFile`	34	24	71%
`HierarchyPropertyParser`	66	48	73%
`HSLColor`	50	42	84%
`ImprovedStreamTokenizer`	84	70	83%
`ImprovedTokenizer`	130	97	75%
`Inflection`	13	10	77%
`IntHashMap`	71	55	72%
`ParameterParser`	68	57	84%
`Range`	152	122	80%
`RationalNumber`	47	40	85%
`SubjectParser`	28	21	75%
`TimeStamp`	32	25	78%
`VCardBean`	173	116	67%
`WeakHashtable`	40	32	80%
`XmlElement`	175	136	78%
`XMLParser`	27	27	100%
Total	**1496**	**1159**	**78%**

The effectiveness rate of our technique is calculated in terms of the number of mutants for which we can infer at least one mutation operator divided by the total number of valid mutants. Overall, we were able to infer 78% of all the considered mutants. This percentage is not consistent throughout all of the programs. However, there is not a single program for which we could not detect the presence of a mutation operator. The class with the least amount of inferred mutation operators was `VCardBean`, with 67% of its mutants classified. The `XMLParser` class is on the

[3]https://github.com/FranciscoRibeiro/morpheus
[4]https://study.code-defenders.org/
[5]https://doi.org/10.6084/m9.figshare.15173934

opposite side with all of its mutants inferred and, therefore, a 100% effectiveness rate. Curiously, the same mutation operator was applied in the same manner by all the players who had to attack this class, i.e., create mutants. This particular mutation occurred in a method that replaces occurrences of the character "&lt" with the character "<". The arguments of the call to the method in question were passed as `string` literals. As such, every call to `replaceAll("&lt;", "<")` was mutated to `replaceAll("<", "<")`. As string literals are considered constants, all these mutations were inferred to be the `ConstantReplacement` operator.

TABLE IV: Inferences per mutation operator

Mutation Operator	#Occurrences
ConstantReplacement	273
RelationalOperatorReplacement	179
VarToVarReplacement	141
ArithmeticOperatorInsertion	105
StatementDeletion	104
NonVoidMethodDeletion	90
VarToConsReplacement	83
ReturnValue	54
UnaryOperatorInsertion	47
ConditionalOperatorReplacement	42
VoidMethodDeletion	40
ArithmeticOperatorReplacement	28
AccessorModifierChange	21
UnaryOperatorReplacement	20
RemoveConditional	20
ConditionalOperatorDeletion	18
ArithmeticOperatorDeletion	16
ConsToVarReplacement	13
MemberVariableAssignmentDeletion	10
ConditionalOperatorInsertion	9
ConstructorCallReplacementNull	7
AccessorMethodChange	6
UnaryOperatorDeletion	5
StaticModifierDeletion	4
ReferenceReplacementContent	4
TrueReturn	4
ArgumentNumberChange	4
BitshiftOperatorReplacement	4
FalseReturn	2
StaticModifierInsertion	2
ArgumentTypeChange	2
BitwiseOperatorReplacement	1
BitshiftOperatorDeletion	1
Negation	1
UNCLASSIFIED	337

We did not infer any mutation operators for 337 of the mutants — Table IV. Ideally, a mutation is a slight syntactic modification that alters the program's behavior. However, sometimes, the generated mutants are not simple modifications because they consist of extensive edits to the source code. These changes are challenging to infer as no mutation operator resembles it. On the other hand, some of these mutants are still small. Nevertheless, they represent intricate code modifications. The following example illustrates such a situation.

```
103c103
<   set(index2, tmp);
---
>   set(index2, get(index2-1));
```

Here, a variable `tmp` got replaced by a method call with different arguments. Many changes are co-occurring, making it difficult to discern the logic behind them.

To get an overview of the entire set of unclassified mutants, we compared every program version for which *Morpheus* did not produce any inference against its original and verified that the difference did not match the criteria of any mutation operator. Nevertheless, we still detected some recurring patterns, shown in Table V. The most frequently detected pattern is adding an instance method call to a variable, which happened 51 times. Adding a statement was the second most spotted type of mutation with 45 occurrences. This is the least specific type of transformation and one of the most difficult to incorporate in mutation testing, as deciding which source code to add to a specific part of a program is not a straightforward task. The third most common mutation pattern was replacing the kind of exception thrown, occurring 21 times. Curiously, we can observe that some patterns displayed in Table V parallel with some of the mutation operators covered by *Morpheus*. For instance, let us consider the fifth most common one, *Overwrite Default Initialization*, which assigns a specific value to a variable, thus not allowing the default ones to occur. This pattern can be seen as the transformation opposing the *MemberVariableAssignmentDeletion* operator (Table I), which eliminates specific assignments to member variables. As another example, if we consider the expression $x < 2$ and then apply the *Negate Expression* pattern, we would get $!(x < 2)$. From another perspective, this is a particular case of rewriting this expression as $x >= 2$, which ends up being covered by the operator *RelationalOperatorReplacement*. The detection of these new patterns has implications regarding mutation-based repair techniques [20, 21, 22, 23, 24, 25]. The candidate fixes for a faulty program are produced by applying mutation operators to suspicious parts of the source code. As such, a repair technique of that kind would not generate an appropriate patch for the cases from which we extracted the patterns reported in Table I. This is because these particular faults originated from applying changes that are not covered by any mutation operator in the literature to the best of our knowledge. Even though the patterns we found in the unclassified cases are used to introduce faults, instead of producing fixes, it is still essential that repair tools incorporate these new operators. As we stated before, some of them revert the effects of already documented operators (*Overwrite Default Initialization* reverts *MemberVariableAssignmentDeletion*). Moreover, the most challenging patches to create are the ones that require adding code [20], which patterns like *Instance Method Call Addition* and *Add String Concat* aim to achieve.

As discussed in Section III-A, sometimes, a mutant can consist of several mutation operators, also called *higher-order mutants* [18], and *Morpheus* can detect these occurrences. Table VI shows the frequency of the number of mutation operators for each program alternative. As we can see by the table, the most common mutants are the ones that get only one mutation operator inferred, totaling 1004 mutants. The program versions with more than one inferred mutation operator combine for 155, representing 10% of all the valid mutants in the repository.

TABLE V: Manual inspection — detected patterns in the 337 program versions with no reported inferences

Pattern	Example	#Occurrences
`Instance method call addition`	var → var**.method()**	51
`Add statement`		45
`Throw exception replacement`	throw new ~~A()~~ **B()**	21
`Replace with new instance`	x = ~~var~~ **new Var()**	20
`Overwrite default initialization`	int x; → int x **= 4**;	20
`Replace Method Call`	var.foo(); → var.**bar()**;	15
`Final keyword removal`	~~final~~ int x;	14
`If block deletion`	~~if(cond) x = 2;~~	11
`Return type change`	public ~~int~~ **long** foo()	9
`Delete statement`		8
`If check deletion`	~~if(cond)~~ x = 2;	7
`Add string concat`	str → str **+ "word"**	7
`Continue/break replacement`	~~continue~~ **break**;	7
`Swap lines`		7
`Primitive to wrapper`	~~int~~ **Integer** x = 2;	7
`Change thrown exception for return`	~~throw new A()~~ **return -1**	6
`Instance change`	var1.foo() → **var2**.foo()	6
`Negate expression`	if(expr) → if(**!**expr)	5
`Delete case`	~~case SOME_VALUE:~~	5
`Change increment size`	i++ → i+=2	4
`Change assigned`	x = 2 → **y** = 2	3
`Delete string concat`	str ~~+ "word"~~	2
`Equivalent default initialization`	Obj x ~~= null~~; → Obj x;	2
`While/If Replace`	while(cond) → **if**(cond)	1
`Variable Type Change`	int var; → **long** var;	1
`Delete Try/Catch`		1
`Change Constant Type`	Integer.MAX → **Long**.MAX	1
`Add Else Block`		1
`Undefined`		50

TABLE VI: Mutation operators per mutant

#Mutation Operators	1	2	3	4	5	16
#Files	1004	132	14	6	2	1

VI. Mutation-Based Repair

We can devise a repair strategy that takes advantage of this new information by translating the bug-inducing changes in terms of mutation operators. Our implementation[6] of such a repair strategy is divided into three parts:

- Extracting fault localization components: interprets the report produced by *Morpheus* and creates components that connect the inferred mutations to their location in the source code;
- Finding nodes in the AST: isolates tree nodes representing source code elements in specific locations;
- Reverting mutations: applies the opposed mutation operator to produce patches.

Figure 9 shows how these parts connect.

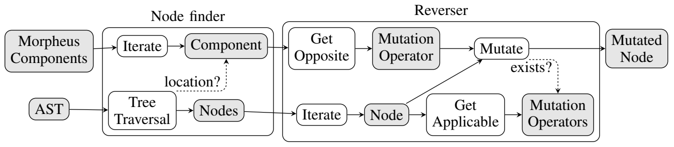

Fig. 9: Repair overview

A. Extracting mutation operators' locations

We can create components that allow us to kickstart the repair process by associating each inferred mutation to its location in the source code. Let us go back to the introductory example. The corresponding component would convey the information in Table VII.

[6]https://github.com/FranciscoRibeiro/auto_repairer

TABLE VII: Mutation's location: introductory example

Mutation Operator	`ArgumentNumberChange`
Callable	`getEnumProperty(Class,String)`
Start-End Old Lines	`132-132`
Start-End Old Columns	`40-61`
Start-End New Lines	`133-133`
Start-End New Columns	`61-74`
Start-End Old Relative Lines	`8-8`
Start-End New Relative Lines	`8-8`

It shows that the *argument number change* mutation operator was inferred in line 133 and spans columns 61 to 74. Furthermore, the transformation was detected in the 8th line of the `getEnumProperty` method. These components can be extracted from the output provided by *Morpheus*.

B. Finding AST nodes

Since these components can pinpoint specific places in the buggy source code, the repair strategy can then analyze the program's AST to find the corresponding nodes.

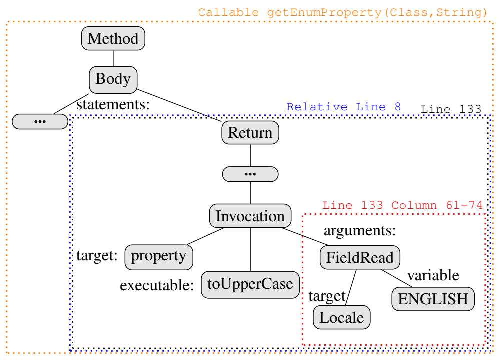

Fig. 10: Finding AST nodes: introductory example

Figure 10 illustrates four different criteria to find nodes in an AST: 1) matching both lines and columns; 2) matching only lines; 3) matching relative lines; 4) matching the callable; As Figure 10 shows, a different number of nodes may be retrieved depending on the selected criteria. As such, it is a matter of deciding on efficiency (1) *vs.* efficacy (2). To increase the likelihood of generating a fix, we should seek to detect many nodes, albeit at the expense of producing a large number of patches. On the other hand, we can limit the number of generated patches by only fetching nodes matching both the lines and columns reported by *Morpheus*.

However, the detected mutations are not always in the same file that we wish to repair. Such would occur when *Morpheus* infers mutations in past file versions, but we wish to use that information and repair a more recent version of a program. In these cases, the reported line numbers may differ from the current location. Thus, it is helpful to use the reported information about the relative line numbers (3) or the callable (4), representing efficiency and efficacy, respectively.

C. Reversing mutations

Following the previous step, the repair process iterates over the returned AST nodes and tries to mutate each one —

Algorithm 2. Every mutation operator that *Morpheus* can infer has another one that performs the opposing transformation. As such, the opposing mutation operator is considered for the inferred mutation in a component (line 1). Then, the strategy retrieves the appropriate mutation operators regarding the node type in question (line 2). Finally, if the opposing transformation belongs to the group of applicable mutations (line 3), it applies it over the AST node (line 4).

Algorithm 2: Algorithm for reversing mutation operators

```
Data: The inferred mutation operator $IMO$; an AST node $N$ that
      we wish to mutate; a mapping of opposing mutation operators
      $MapO$; a mapping between types of AST nodes and their
      applicable mutation operators $MapN$; an empty list of
      mutated nodes $MutN$
Result: The list of mutated nodes $MutN$
1 opposite ← $MapO[IMO]$; // get opposite mutOp of $IMO$
2 mut_ops ← $MapN[N]$; // get mutOps applicable to $N$
3 if opposite ∈ mut_ops then
4     $MutN$ ← repair(opposite, $N$); // apply opposite to
          node $N$
5 end
```

VII. Case Studies with Real Bugs

The main idea we want to deliver is that the *semantics* behind a *bug* can guide the repair process of a program. To show this, we used *Morpheus* to analyze *Bugswarm* and *Defects4J*, extracting several case studies from real-world programs. Furthermore, we also implemented an automated repair process that successfully fixed all the studied programs. These experiments are available for replication[7,8].

a) Bugswarm: We selected case studies from this repository for which a simple fix, susceptible of being represented by a mutation, makes the program pass every test. We looked back through the commit history of each selected case study and searched for the most recent commit that began failing these tests. There, we detected mutations opposing the bug-fixing modification, likely responsible for introducing the bug.

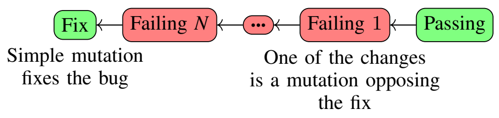
Fix
Failing N
•••
Failing 1
Passing
Simple mutation fixes the bug
One of the changes is a mutation opposing the fix

Fig. 11: Selection criteria for the case studies - *Bugswarm*

b) Defects4J: In this repository, the fixes for the selected case studies can also be obtained by performing a simple mutation. In *Defects4J*, however, the commit in which we detect the opposing transformation does not necessarily cause the software to fail its tests. Moreover, the file that makes the tests fail may have changed multiple times since introducing the original reason for the bug.

[7]https://github.com/FranciscoRibeiro/bugswarm-case-studies
[8]https://github.com/FranciscoRibeiro/d4j-case-studies

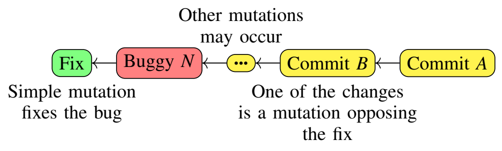
Other mutations may occur
Fix
Buggy N
•••
Commit B
Commit A
Simple mutation fixes the bug
One of the changes is a mutation opposing the fix

Fig. 12: Selection criteria for the case studies - *Defects4J*

Besides repairing all analyzed programs, when guiding the repair process, our approach has four key advantages over spectrum-based fault localization (SFL) reports: 1) **Efficiency:** *Morpheus* detects fewer mutants when compared against the many lines SFL reports. Moreover, SFL sometimes misses to rank the faulty line in which *Morpheus* can spot a mutant; 2) **Compilation:** buggy programs may fail to compile as *Morpheus* does not need the program's execution trace, whereas SFL would not generate a report; 3) **Reachability:** mutations can be detected in any commit of a program's history, regardless of whether there are failing tests or not; 4) **Granularity:** inferred mutants have more granularity than the line number provided by SFL, allowing program repair to use different criteria (recall Figure 10).

Due to lack of space, we provide a detailed explanation of the case studies separately on the following page:

https://github.com/FranciscoRibeiro/qrs21-case-studies-report

VIII. Threats to Validity

Our work has two main objectives. Firstly, to assess whether we can translate the evolution of a program in terms of mutation operators. More precisely, are the changes applied to a program equivalent to the application of well-known mutation operators? Secondly, to check if the information about inferred mutations can benefit automated program repair. That is, can programs be fixed more efficiently by using this new knowledge over regular SFL reports?

As we have shown, the answer to the previous questions is yes. There are, however, several aspects that may affect the validity of our work.

a) ***Internal Validity****:* There is no guarantee that the analyzed programs compare equally in terms of susceptibility to mutations. Mutations are slight syntactic modifications that alter the program's semantics, and, as such, the examined mutants may consist of more complex changes which do not correspond to any documented mutation operators. Furthermore, the mutants from *CodeDefenders* were created by people learning about the topic in question. Although some mutants may not convey the desired simple nature, we think our results show that a considerable part of them do hold to this standard.

b) ***External Validity****:* The mutants from the *CodeDefenders* repository were created with the explicit goal of producing faults. Nonetheless, we showed that our work generalizes to a real-world context by using the inferred information to repair open-source projects from the *Bugswarm* and the *Defects4J* repositories in which faults were unintentionally introduced. Quoting DeMillo et al. [10] on a quality about programmers: "they create programs that are close to being correct!".

*c) **Construct Validity**:* The *CodeDefenders* mutants were produced by subjects who were aware that they were to be used in research. We do not believe this to have compromised our measures because the origin of the repository is independent of our study, and the intentions of inferring mutation operators were never communicated. Moreover, our analysis of real-world faults further strengthens this point as programs were developed in a completely disconnected context disassociated from any research intentions.

*d) **Conclusion Validity**:* The idea transmitted to *CodeDefenders* players was they should replicate the behavior of mutation operators. Still, some mutants did not obey this practice. We conclude that real-world program changes can be described in terms of mutation operators, as demonstrated by the reported real-world case studies. Furthermore, previous studies have already shown associations between real faults and mutation operators.

IX. Related Work

Jia and Harman [18] present the concept of *higher-order mutation testing*, in which mutants are not individual faults but are composed of several faults. They emphasize *subsuming higher-order mutants*, which are notably hard to kill. The program versions for which *Morpheus* infers two or more mutation operators are instances of higher-order mutants.

Debugging is one of the most expensive actions in the development cycle [26] and a considerable effort is put into fault localization [27, 28]. *MUSE* [29] applies mutations to both faulty and correct statements to rank the most suspicious lines, improving over previous state of the art. The reasoning is that tests that pass in the original program are more likely to fail when correct statements are mutated and less likely to do so when faulty lines are mutated. Mutation-based fault localization has also been applied to projects written in multiple programming languages, reporting high accuracy and proposing new mutation operators [30]. Other approaches [31, 32] detect suspicious statements by calculating similarities between mutants. Zhang et al. [33] describe a technique in which artificially produced mutants are mapped to higher-level programming edits. Instead, we provide more granularity by interpreting structural changes to infer what mutation operator is being applied. Fault localization lacks the semantics behind the faults it spots. *Morpheus* can provide this as it computes the context of the modifications it detects.

Mutation-based program repair [20, 21, 22, 23, 24, 25] uses fault localization to mutate the most suspicious lines. A mutant is considered as a potential fix if it passes all the test cases. If effectiveness is the focus, a large set of mutation operators should be considered to cover the largest number of faults. On the other hand, techniques aiming for efficiency should only apply a small set of mutation operators to minimize overhead, sacrificing the ability to fix some types of faults. *Morpheus* infers mutation operators and repair strategies can take advantage of this semantics to apply modifications that revert the faulty effects. Tan and Roychoudhury [34] aim to repair regression errors by manually extracting fix patterns from a project's history and applying them to suspicious statements. Our approach differs, as the inference process of our tool automatically detects the application of mutation operators. Moreover, we aim to infer operators used by mutation testing tools, instead of high-level transformations such as "Revert to previous statement". The automatic detection of bug fixes [35] is also based on an established taxonomy, with changes being analyzed at the *AST* level. However, the list of considered bug fixes — 25 patches — is more generic and not as extensive as ours — 34 mutations.

Generating test cases through mutations [36] has been applied in web page testing [37], and tools like *Sapienz* [38] are already following this approach and successfully detecting bugs in mobile applications used by millions of people.

Tree differencing has been applied to build files [39]. Hence it is well suited to address a project's configuration. Our work focuses on a much broader aspect, requiring the *ASTs* of source code to reason about the issues.

Some approaches [40] mine project repositories to find frequent bug patterns for languages that research has yet to report mutation operators. Our work differs, as we present a tool that automatically detects the application of well-documented mutation operators to a correct program.

X. Conclusions and Future Work

We presented an inference technique that defines the context behind source code changes by associating them with well-known mutation operators. We implemented it as the *Morpheus* tool and analyzed several manually modified programs. Our results show that this is a sound approach, as we were able to infer mutation operators for 78% of the 1496 valid mutants in *CodeDefenders* and that 10% of these are *higher-order mutants* [18]. Furthermore, we have also analyzed several case studies extracted from real-world projects in *Bugswarm* and *Defects4J*, showing the benefits of our approach regarding automated program repair. We fixed these programs by implementing a repair tool that reverts bug-introducing changes based on Morpheus' information by applying the opposing mutation. The concept of *higher-order mutants* was essential, as highlighted by the case studies. The repair strategy focused on fixing the effect of a single atomic mutation to create a patch for a program that was modified in separate places by different mutation operators. Nonetheless, there are still faults which need more extended and more intricate patches. As such, we wish to explore the use of multiple mutations to repair programs, either by compounding them to build more complex expressions or by applying them at separate places to patch independent modifications.

Replication Package

All the necessary resources to replicate this study, as well as the full set of results, are publicly available:

- **Mutant repository:** *https://study.code-defenders.org/*
- **Morpheus:** *github.com/FranciscoRibeiro/morpheus*
- **Repair tool:** *github.com/FranciscoRibeiro/auto_repairer*
- **Dataset:** *https://doi.org/10.6084/m9.figshare.15173934*
- **Bugswarm case studies:** *github.com/FranciscoRibeiro/bugswarm-case-studies*
- **D4J case studies:** *github.com/FranciscoRibeiro/d4j-case-studies*
- **Case studies report:** *github.com/FranciscoRibeiro/qrs21-case-studies-report*

ACKNOWLEDGMENTS

Francisco Ribeiro and João Saraiva would like to thank the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia through reference UIDB/50014/2020. Francisco Ribeiro would also like to acknowledge a PhD scholarship with reference SFRH/BD/144938/2019. Rui Abreu would like to acknowledge FCT through reference UIDB/50021/2020, the SecurityAware Project (ref. CMU/TIC/0064/2019) - also funded by the Carnegie Mellon Program, and the FaultLocker Project (ref. PTDC/CCI-COM/29300/2017).

REFERENCES

[1] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019.

[2] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class mutation operators for java," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*. IEEE Press, 2002.

[3] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, "The class-level mutants of mujava," in *Proceedings of the 2006 International Workshop on Automation of Software Test (AST'06)*. ACM, 2006.

[4] A. Derezińska, "Advanced mutation operators applicable in c# programs," in *Software Engineering Techniques: Design for Quality*. Springer US, 2007.

[5] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, 2014.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings. 27th International Conference on Software Engineering (ICSE'2005)*, May 2005.

[7] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'96)*. ACM.

[8] A. S. Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*. Association for Computing Machinery, 2011.

[9] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 2016.

[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, 1978.

[11] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Softw. Pract. Exper.*, vol. 21, no. 7, pp. 685–718, 1991.

[12] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: A mutation system for java," in *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, 2006.

[13] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM, 2014.

[14] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*. ACM, 1996.

[15] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[16] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. ACM, 2014.

[17] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," *SIGPLAN Notes*, 2008.

[18] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, 2009, proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM'08).

[19] J. M. Rojas, T. D. White, B. S. Clegg, and G. Fraser, "Code defenders: Crowdsourcing effective tests and subtle mutants with a mutation testing game," in *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE Press, 2017.

[20] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST'10)*. USA: IEEE Press, 2010.

[21] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, 2016.

[22] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. ACM, 2019.

[23] B.-C. Rothenberg and O. Grumberg, "Sound and complete mutation-based program repair," in *Formal Methods (FM'16)*, J. Fitzgerald, C. Heitmeyer, S. Gnesi, and

A. Philippou, Eds. Springer International Publishing, 2016.

[24] V. Debroy and W. E. Wong, “Combining mutation and fault localization for automated program debugging,” *Journal of Systems and Software*.

[25] U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. Di Guglielmo, G. Pravadelli, and F. Fummi, “Combining dynamic slicing and mutation operators for esl correction,” in *2012 17th IEEE European Test Symposium (ETS)*, 2012.

[26] I. Vessey, “Expertise in debugging computer programs: A process analysis,” *International Journal of Man-Machine Studies*, 1985.

[27] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA’11)*. ACM, 2011.

[28] A. Ang, A. Perez, A. v. Deursen, and R. Abreu, “Revisiting the practical use of automated software fault localization techniques,” in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2017, pp. 175–182.

[29] S. Moon, Y. Kim, M. Kim, and S. Yoo, “Ask the mutants: Mutating faulty programs for fault localization,” in *Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST’14)*, 2014.

[30] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, “Mutation-based fault localization for real-world multilingual programs (t),” in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE’15)*, 2015.

[31] M. Papadakis and Y. Le Traon, “Using mutants to locate "unknown" faults,” in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST’12)*. IEEE Press, 2012.

[32] ——, “Metallaxis-fl: Mutation-based fault localization,” *Software Testing Verification Reliabiability*, vol. 25, no. 5–7, 2015.

[33] L. Zhang, L. Zhang, and S. Khurshid, “Injecting mechanical faults to localize developer faults for evolving software,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA’13)*. ACM, 2013.

[34] S. H. Tan and A. Roychoudhury, “Relifix: Automated repair of software regressions,” in *Proceedings of the 37th International Conference on Software Engineering (ICSE’15)*. IEEE Press, 2015.

[35] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia, “Towards an automated approach for bug fix pattern detection,” *arXiv preprint arXiv:1807.11286*, 2018.

[36] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, March 2012.

[37] S. Almeida, A. C. R. Paiva, and A. Restivo, “Mutation-based web test case generation,” in *Quality of Information and Communications Technology*, M. Piattini, P. Rupino da Cunha, I. García Rodríguez de Guzmán, and R. Pérez-Castillo, Eds. Springer International Publishing, 2019.

[38] K. Mao, M. Harman, and Y. Jia, “Sapienz: Multi-objective automated testing for android applications,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 94–105.

[39] C. Macho, S. Mcintosh, and M. Pinzger, “Extracting build changes with builddiff,” in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR’17)*. IEEE Press, 2017.

[40] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, “Discovering bug patterns in javascript,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE’16)*. ACM, 2016.