Correcting Instruction Expression Logic Errors with GenExp: A Genetic Programming Solution

Mohammed Bekkouche

Abstract

Correcting logical errors in a program is not simple even with the availability of an error locating tool. In this article, we introduce GenExp, a genetic programming approach to automate the task of repairing instruction expressions from logical errors. Starting from an error location specified by the programmer, we search for a replacement instruction that passes all test cases. Specifically, we generate expressions that will substitute the selected instruction expression until we obtain one that corrects the input program. The search space is exponentially large, making exhaustive methods inefficient. Therefore, we utilize a genetic programming meta-heuristic that organizes the search process into stages, with each stage producing a group of individuals. The results showed that our approach can find at least one plausible patch for almost all cases considered in experiments and outperforms a notable state-of-the-art error repair approach like ASTOR. Although our tool is slower than ASTOR, it provides greater precision in detecting plausible repairs, making it a suitable option for users who prioritize accuracy over speed.

Keywords: error correction, instruction expression, plausible patch, crossover, mutation.

MSC 2020: 68W50, 68T05, 68T20, 68N30.

1 Introduction

Developers spend a significant amount of their engineering time and effort in finding and fixing bugs in their code [1]. Even after locating errors, program debugging remains a challenging task. Logical errors

©2023 by Computer Science Journal of Moldova
doi:10.56415/csjm.v31.12

are among the most common bugs in programming. The programmer, by correcting these errors, in a totally manual way, can insert others which make matters worse. For this, we need to use an automatic or at least a semi-automatic approach for error correction. Many researchers have dealt with this recent subject in view of its importance in the process of software development. Among the early proposed approaches, we can cite [2] and [3], which were the pioneers in utilizing the genetic programming (GP) technique to evolve the erroneous program until finding an individual that resolves the errors. The problem which can oppose the use of this technique in the context of error correction is that the number of individuals of generated programs can become important before arriving at a solution. To address this, we propose an approach called GenExp in this paper. GenExp leverages knowledge about the program being repaired to find plausible patches for faulty instruction expressions.

The automatic phase of this approach follows a series of steps that form the structure of the genetic algorithm:

1. create the initial population,
2. evaluate each candidate using a fitness function,
3. evolve new generation,
4. evaluate candidates,
5. if a candidate that corrects the program, according to the test cases considered as input, is found, the process ends. Otherwise, we need to repeat steps 3 and 4.

The goal is to select the best individuals and form other individuals that combine elements from the previous stage. The process starts with a set of individuals randomly generated and enriched with expressions from the program under repair. To create the initial population, our basic algorithm proposes randomly generating expressions (candidates or patches) from a set of variables, constants, and operators chosen by the user. We propose that these sets be defined automatically based on the program to be repaired and the erroneous instruction. The process of GP from this initial population can lead to an explosion in the number of generated individuals before reaching a plausible patch. To reduce the effect of this problem, we adopt the idea presented by the GenProg approach [3] which suggests that most defects can be repaired by adopting existing code from elsewhere in the program. This is referred to as ”repair ingredients” or ”patch ingredients”. In GenProg, a candidate patch is synthesized from instructions taken as is elsewhere in the application. Our idea which is close to that of CARDUMEN [4] is to reuse code instruction expressions as ingredients rather than reusing raw, unmodified code elements (such as raw instructions in GenProg). For this, we add to the initial population the expressions of instructions of the program which allow it, by replacing the erroneous expression, to compile correctly. We calculate the degree of correctness of the candidates according to the input test cases and this will be our fitness function. To evolve the current generation into a new generation, we first select the best individuals in terms of their fitness function value to survive in the new population. Then, we apply a set of mutations and crossovers to individuals in the current population to generate the other individuals of the new generation. The goal is to improve the expressions. We continue evolving new generations until we obtain a plausible patch, that is to say a patch that produces correct outputs for all inputs in the test suite [5].

We evaluated the capacity of GenExp to discover plausible patches (test-suite adequate patches). Our tool was implemented in Java to repair Java programs. To do the experiments, we constructed a set of erroneous academic programs. Each of these programs contains a single error to be corrected. We compared our implementation with three approaches from the ASTOR tool [6], an open-source framework for repairing buggy Java program. The results showed that our tool successfully found a plausible patch for nearly all the programs under repair and outperformed the ASTOR approaches, even when united[1]. However, our tool is slower than other tools. Nonetheless, thanks to its higher precision in detecting plausible repairs, GenExp can be a

[1]To harness the collective power of ASTOR’s approaches, it is sufficient for at least one of them to discover a plausible patch in order to claim that ASTOR has fixed the program.

suitable option for users who prioritize accuracy[2] over speed.

This article is structured as follows. We illustrate our approach on an example in Section 2. Section 3 deals with the presentation of relevant works that have a strong interest in the context of error correction. The details of the approach proposed and results are exposed respectively in Sections 4 and 5. The conclusion is presented in Section 6.

2 Illustrative example

We consider an erroneous version of SquareRoot program (Listing 1). The SquareRoot correct program finds the square root integer part of an integer value greater than or equal to 0. If we assume that the locating errors phase is complete and so on we get a set of suspicious instructions. We obtain this set thanks to an error localization tool. Now comes the role of the programmer to analyse the produced set and choose the instruction that should be corrected using GP. In fact, we aim to modify the expression of this instruction until the program becomes correct. The programmer after checking realizes that the instruction in the line 10 ($res = i + 1$) should be corrected. The error correction tool then evolves the program by just changing the expression of this instruction until the correct program is obtained. The tool changes the expression of the selected instruction, whether it is an assignment or a branching. A timeout is specified to indicate that the tool has not found solutions. In such cases, the programmer will review the selected instruction and choose another one.

Listing 1. The SquareRoot program with an error

```
1   public class SquareRoot {
2           public static int squareRoot (int val) {
3                   int i = 1;
4                   int v = 0;
5                   int res;
6                   while (v < val){
7                           v = v + 2*i + 1;
```

[2]One program repair tool can be considered more accurate than another when it consistently achieves a higher success rate in finding plausible patches when benchmarking erroneous programs.

```
8                   i = i + 1;
9               }
10              res = i+1; //error: should be res = i-1
11              return res;
12          }
13      }
```

Here we will explain how to automatically search for a plausible patch for a specific suspicious instruction. First, the programmer determines which operators, variables, and constants to take, these sets allow to build expressions that replace "i+1" in "res = i+1". This stage is very important, wrong selection may result in making the correct expression inaccessible. We suppose that these sets are $\{+, -, *\}$ (for operators) and $\{val, i, v\}$ (for variables) and integers in the interval $[0, 10]$ (for constants). Here, these sets are defined manually. However, in the subsequent Subsection 4.1, we will explore improvements that enable the automatic construction of these sets from the program under repair. So the algorithm continues by applying the following genetic processes from an initial population of individuals: selection, crossover, and mutation. To establish a preference order among individuals, a fitness function is utilized, which calculates the number of successful executions in the modified program (of course test cases should be used with at least one counterexample). An expression that passes all test cases is considered a potential correction (plausible patch). Table 1 prints the output expected for each test case input used.

To illustrate how the selection of individuals works, we display the computed outputs for the same inputs while considering the expression "(v-val)" instead of "i+1" in the statement at line 10 (Table 1, column 3). The executed program is depicted in Listing 2.

Listing 2. SquareRoot program, considering the expression "(v-val)" in the statment at line 10.

```
1   public class SquareRootv2 {
2           public static int squareRoot (int val) {
3                   int i = 1;
4                   int v = 0;
5                   int res;
6                   while (v < val){
7                           v = v + 2*i + 1;
8                           i = i + 1;
9                   }
```

```
                    res = (v-val);
            }
    }
```

Table 1. Computed and expected outputs for SquareRoot and SquareRootv2

Input (val)	Output for SquareRoot	Output for SquareRootv2	Expected output
16	6	8	4
20	6	4	4
25	7	10	5
30	7	5	5
36	8	12	6
40	8	8	6
49	9	14	7
60	9	3	7
64	10	16	8
80	10	0	8
81	11	18	9
90	11	9	9
100	12	20	10

This result shows that there are only two successful test cases (input = $\{20, 90\}$), while the others do not produce the expected output. Based on this, we can conclude that "(v-val)" is not a plausible patch.

The initial population consists of a set of candidates (expressions) that were randomly generated $(6, 8, 1, v, v + 2, 2 * val, i * v, i * val, ...)$. Each candidate in this set will be evaluated using the same method as illustrated previously. In order to generate a new generation, we apply genetic operators to candidates. For example:

- i*val and 1 crossover to i*1
- i*1 mutates to i-1

The iterative process of generating new generations continues until a termination condition is met:

1. An expression that successfully executes all test cases (a plausible patch) has been found;
2. The specified timeout has elapsed.

In our case, the first condition is checked because the expression ”i-1” permits obtaining all expected outputs. The programmer analyses this result to decide if it should be considered definitive (a patch is considered definitive when the programmer finds it to be correct). This patch is a correction because it achieves not only the test cases used but also all the functionalities of SquareRoot program. The erroneous instruction will become ”res = i - 1”.

3 State of the art

3.1 Categories of patch generation techniques

Patch generation techniques can be categorized into four main classes [7]: heuristic-based, template-based, constraint-based, and learning-based repair techniques.

3.1.1 Heuristic-based repair techniques

Heuristic search methods use a generation-and-test methodology, building and iterating over a search space of syntactic program modifications [8]. Among these methods, GenProg [9] is regarded as a seminal work in this field. It utilizes GP to evolve variants of the program until one is found that both retains the required functionality and also avoids the defect in question. The technique takes as input a program, a set of successful positive test cases that encode the required behaviour of the program, and a failed negative test case that demonstrates a defect. GenProg defines a fitness function that measures the quality of each program variant based on the number of passing and failing test cases. The search is restricted to only produce changes based on structures in other parts of the program. Mutation and crossover genetic operations only operate on the region of the program that is relevant to the error, i.e., the parts of the program that were on the path of execution that produced the error. Arcuri and Yao [2] are the ones who proposed the idea of using GP to co-evolve programs and unit tests in order to automate the task of fixing bugs. Subsequently, Arcuri [10] developed a research prototype called JAFF, which models bug fixing as a research problem. RSRepair (Random-Search-based Repair) [11] is a tool that repairs program defects using the same mutation operations as GenProg, but it employs random search instead of GP. Unlike GP, which requires an evaluation of the fitness of a candidate patch even if GenProg has been aware that the patch is invalid, random search has no such constraint. Furthermore, RSRepair can speed up the process of early identification of invalid patches using traditional test case prioritization techniques. SCRepair (Similar Code-based Repair) [12] uses the same mutation operators as RSRepair to modify the faulty program. Note that the insertion operator needs code from other places, which is the main difference between SCRepair and RSRepair. To select a new code to replace the existing code during the mutation process, SCRepair introduces a metric that calculates the similarity between two code fragments based on their Abstract Syntax Tree (AST). The most suitable instruction is chosen to replace the faulty location, and test cases are used to verify the elimination of the bug. In order to improve the efficiency of research via GP for program repair, Yuan and Banzhaf [13] present a new repair system based on this technique for automated repair of Java programs, called ARJA. ARJA is mainly characterized by a new patch representation, a multi-objective search, a test filtering procedure, and several strategies to reduce the search space.

3.1.2 Template-based repair Techniques

An automated program repair strategy involves generating concrete patches based on remediation templates, also known as program transformation patterns. This strategy is widely used in the literature and has been implemented in several automated program repair systems [14]. Techniques like GenProg, which rely on heuristics, can generate nonsensical patches due to the randomness of their mutation operations. To address this limitation, a new patch generation approach called Pattern-based Automatic program Repair (PAR) [15] has been proposed. PAR utilizes patch templates learned from existing human-written patches. Durieux, Cornu, Seinturier, and Monperrus [16] proposed NPEfix, a new technique to explore the search space of potential fixes for null pointer exceptions using meta-programming. NPEfix is based on nine predefined fix templates specifically tailored for null pointer exceptions. Long, Amidon, and Rinard designed Genesis [17] to infer remediation patterns from successful human fixes for three types of defects: null pointer, out-of-bounds, and class cast. Genesis leverages the expertise and patching strategies of developers around the world to automatically fix bugs in new applications. Stack Overflow has millions of posts that could potentially be useful for fixing numerous bugs. This observation motivates Liu and Zhong [18]'s work on extracting repair patterns from Stack Overflow for automatic program repair. To find as many adequate fixes as possible for a test suite for a given bug, Martinez and Monperrus [4] created CARDUMEN, an automated repair approach based on extracted patterns that has ultra-large search space. CARDUMEN extracts code patterns from code being repaired. Liu, Koyuncu, Kim, and Bissyandé [14] implemented TBar, an automated patch generation system that incorporates a superset of patch patterns collected, summarized, organized, and labelled from literature data.

3.1.3 Constraint-based repair techniques

Typically, these approaches infer semantic constraints from the provided test cases and then generate the appropriate test suite fix by solving the resulting constraint satisfaction problem, in particular, the SMT problem [13]. Nguyen, Roychoudhury, and Chandra [19] proposed SemFix, a pioneering tool for constraint-based program repair. Given a program location to be fixed, constraints on the expression to appear in the program location are derived so that the modified program passes all the given tests. The repair constraints are generated by symbolic execution and the expression to be repaired is obtained by program synthesis. Ke, Stolee, Le Goues, and Brun [20] developed a repair method based on semantic code search called SearchRepair. The idea is to utilize semantic code search [21] on existing open-source code to find correct implementations of buggy components and methods and use the results to automatically generate fixes for defective software. This method encodes a database of human-written code fragments as SMT constraints on input/output behaviour and searches the database for potential fixes with an input/output specification. Mechtaev, Yi, and Roychoudhury [22] present a new semantic-based repair method called DirectFix that generates the simplest fix to maximize the preservation of the program structure of the buggy program. To take into account repair simplicity in an efficient way, their method merges fault localization and repair generation into a single step. They achieve this by leveraging partial MaxSAT constraint solving and component-based program synthesis. The same authors proposed Angelix [23], a new semantic-based repair method that is adaptable to programs of similar size as heuristic-based repair tools like GenProg. They demonstrate that Angelix is more scalable than previously proposed semantic-based repair methods such as SemFix and DirectFix. The scalability of Angelix is attributed to the new lightweight repair constraint called angelic forest, which is independent of the size of the program being repaired. Furthermore, this repair method can repair multiple buggy locations that depend on each other. In [24], the authors investigate automated error repair using a reference implementation. They propose deriving a correct specification from the reference implementation and using it to guide the repair of the program to solve the test overfitting problem.

3.1.4 Learning-based repair techniques

Machine learning techniques can enhance the efficiency of automatic bug-fixing systems. Unlike the techniques in the aforementioned three categories, learning-based techniques typically require additional training data (i.e., the tuples of buggy, context, and fixed lines of code) to capture the intricate relationships between buggy and fixed code [7]. For example, [25] presents an algorithm that learns model parameters through a training set of successful human patches collected from open-source project repositories. It generates a candidate patch space, utilizes the model to rank the candidate patches in order of likely correctness, and validates the ranked patches against a suite of test cases to discover the correct patches. Tufano, Watson, Bavota, Di Penta, White, and Poshyvanyk [26] extensively evaluate the ability of adopting neural machine translation (NMT) techniques to learn code fixes from real bug fixes. Furthermore, Lutellier, Pham, Pang, Li, Wei, and Tan [27] employ ensemble learning on the combination of convolutional neural networks and a new context-aware NMT architecture to automatically fix bugs in multiple programming languages. This architecture separately represents buggy source code and its surrounding context. SequenceR [28] utilizes sequence-to-sequence learning on source code to generate one-line patches. This approach employs the copy mechanism to address the challenge of unlimited vocabulary in the source code. In [29], the authors aim to advance deep learning-based automated program repair by introducing DEAR, a deep learning-based model that facilitates fixing general bugs with changes dependent on one or more buggy statements belonging to one or multiple buggy hunks of code.

3.2 The ASTOR tool

The ASTOR tool, also known as Automatic Software Transformations for Program Repair (ASTOR) [6], automatically repairs Java programs. We utilized this tool in our experiments. It takes a buggy program, its test suite with at least one failed test case as input, and generates a patch, if possible, that fixes the bug (i.e., all test cases pass after the repair). The change point is driven by an existing spectrum-based fault localization technique called Ochiai [30]. ASTOR offers various modes, each corresponding to a different repair algorithm (their original implementations were for other programming languages). The modes we employed in our experiments are the following: JGenProg, MutRepair, and CARDUMEN.

JGenProg JGenProg is a Java implementation of GenProg [3]. It operates at the statement level, meaning it deletes, replaces, and inserts statements. The inserted code fragments through addition or replacement always originate from the same program. The replacement operator replaces one statement with another of the same type (e.g., an assignment is only replaced by another assignment). There is a risk that this Java implementation may not reflect the actual performance of the original GenProg system for C [31].

MutRepair MutRepair implements the approach of Debroy and Wong [32], which is a repair approach that applies operators from mutation testing to repair C code. MutRepair applies mutation operators to suspicious ”if” condition statements and performs a single change to the condition. There are three types of mutation operators: relational (there are six interchangeable operators: $>$, $>=$, $<$, $=<$, $==$ and $!=$); logic (there are two: OR, AND); unary (there are two mutations: *negation* and *positivation*[3]).

CARDUMEN CARDUMEN is a repair approach based on mined templates that has an ultra-large search space [4]. It extracts code templates from the application being repaired to create a template-based search space. The repair always consists of replacing the suspected code element with an instance of the code template.

Note that MutRepair and CARDUMEN are template-based approaches, while JGenProg is heuristic-based.

4 Approach

The automatic stage of our approach (see Algorithm 1) takes as input the following: an erroneous program (*prog*), error provided from an error location tool (such as LocFaults [33], [34] or BugAssist [35], [36]) and examined by the programmer (*error*), failing and successful test cases (*tests*) with expected output for each one, and a timeout for process execution (*timeOut*). It automatically generates a set of potential corrections for *error* in *prog* that passes all tests in *tests*. We perform, for this, the GP algorithm in five steps: Initial population, Fitness function, Selection, Mutation, and Crossover.

The process begins with a set of individuals called *population* (Alg. 1, line 2). Each individual (expression) represents a possible correction randomly generated from sets that contain *variables*, *constants*, and *operators* provided from the programmer. For example, $i * val$ is an expression that can be generated for this configuration: $variables = \{..., i, ..., val, ...\}$, $operators = \{..., *, ...\}$, and

[3]Removal of the negation operator.

constants = {...}.

In the next step, we compute a fitness score for each expression in *population* to measure how these expressions correct the instruction error in *prog* (Alg. 1, line 3). If *population* contains at least one individual that succeeds all test cases, the error correction algorithm ends and returns all individuals with higher fitness function (Alg. 1, lines 10–11). In the opposite case, we need to build new generations of populations using crossover and mutation genetic operators until we find one or more individuals that satisfy the above condition, and then return them to the programmer (Alg. 1, lines 4–9). Algorithm 1 can terminate with no correction found in case that the execution exceeds the specified timeout (Alg. 1, lines 12–14).

Algorithm 1 Algorithm for error correction from a suspected instruction.

Inputs: *prog*: the erroneous program; *error*: the error to correct; *tests* : a map that stores the expected output for each test case input; *timeOut*: the maximum allowed execution time.
Output: a set of potential corrections for error to be checked by the programmer.

```
1: (variables, constants, operators) ← read(); {Read variables, constants, and operators from the user or programmer.}
2: population ← create(variables, constants, operators, size) {Build the initial population.}
3: individualsFitness ← fitnessFunction(population, prog, error, tests); {Compute the fitness score for each individual in population.}
4: while (! existSolution(individualsFitness) and ! exceed(timeOut)) do
5:     newPopulation ← selectBestIndividuals(population, individualsFitness, $ratio_1$);
6:     newPopulation ← newPopulation ∪ mutation(population,$ratio_2$);
7:     newPopulation ← newPopulation ∪ crossover(population,$ratio_3$);
8:     individualsFitness ← fitnessFunction(newPopulation, prog, error, tests);
9: end while
10: if existSolution(individualsFitness) then
11:     return bestIndividuals(population, individualsFitness);
12: else
13:     print("No plausible patch found, the timeout indicated is elapsed.");
14:     return null;
15: end if
```

existSolution(*individualsFitness*): tests if it exists an element in *individualsFitness* equal to the number of test cases in *tests*, this means that the current generation includes at least one potential correction.
exceed(*timeOut*) : tests if the timeOut is exceeded.
bestIndividuals(*population*) : selects individuals with highest fitness score in *population*.

The idea behind the composition of a new generation is to develop new expressions that may contain a plausible patch or, at the very least, improve the fitness score of individuals from the previous generation (Alg. 1, lines 4–9). To achieve this, we apply a set of crossovers and mutations to the individuals in the current population. However, before doing so, we select the fittest individuals and include them in the next generation. A mutation modifies an individual to generate a new one. We perform mutations at a specific point, which means that a random element in the expression to be mutated will be replaced with a new element from the same category (variables, operators, or constants). A crossover combines individuals to produce a new expression which may improve their fitness score. To accomplish this, we replace a randomly selected sub-expression in the first individual with a randomly selected sub-expression from the other individual. Figure 1 [37] illustrates an example of how subtree crossover can be used to combine two tree structures to create a new one. To specify how many times these genetic operations (selection, mutation, crossover) are activated, we use ratio values for each one (*ratio*$_1$, *ratio*$_2$, and *ratio*$_3$).

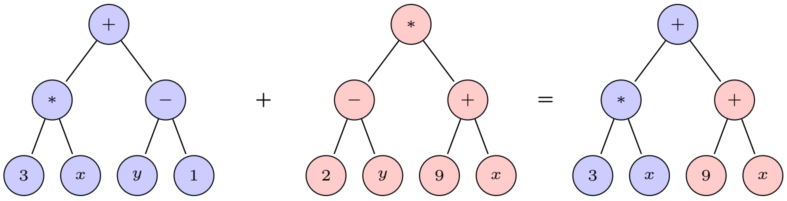

Figure 1. Example of subtree crossover operation

To calculate the fitness score for each expression in the population (*population*), we generate a program for each individual by putting it in the place of the error expression in *prog*, *prog*′ is used to denote the modified program (Alg. 2, line 2). *prog*′ will be compiled (Alg. 2, line 3) to obtain *compiledProg*. Next, we execute *compiledProg* on each test case input in *tests*, comparing the obtained results with the expected outputs to determine the number of successful executions (Alg. 2, lines 8–11). This count represents the fitness score for the current individual's expression and is stored in a list. Once the fitness scores for all individuals have been computed, this list, denoted by the variable *individualsFitness* in Algorithm1, will be returned (Alg. 2, line 15).

Algorithm 2 fitnessFunction: Function to compute fitness score for individuals.

Inputs: *population* : individuals ; *prog* : program to be corrected; *error*: instruction to be corrected in *prog*; *tests*: test cases.
Output : a list of fitness values.

```
 1: for all $individual \in population$ do
 2:     $prog' \leftarrow$ replaceError($prog$, $individual$, $error$); {Replace the error expression in $prog$
        with $individual$, and let $prog$ be the modified program.}
 3:     $compiledProg \leftarrow$ compile($prog'$); {Compile $prog'$.}
 4:     $testCases \leftarrow$ getTestCases($tests$); {Get test cases from $tests$.}
 5:     $cpt \leftarrow 0$;
 6:     for all $testCase \in testCases$ do
 7:         $output \leftarrow$ run($compiledProg$,$testCase$); {Execute $compiledProg$ by using
            $testCase$ as input.}
 8:         $expectedOutput \leftarrow$ getExpectedOutput($testcase$, $tests$); {Retrieve the expected
            output of $testCase$ from $tests$.}
 9:         if Equal($output$,$expectedOutput$) then
10:             increment($cpt$);
11:         end if
12:     end for
13:     add($cpt$,$result$);
14: end for
15: return $result$;
```

Equal(*output*,*expectedOutput*) : tests if data in *output* is equal to that in *expectedOutput*.
increment(*cpt*) : increments *cpt*.

4.1 Improvements

Our error repair algorithm can be improved. The improvements we propose are related to the calculation of the fitness function and the initial population.

Fitness function To repair an instruction expression of a program that produces a numeric value as output, we can consider the following improvement related to the calculation of the fitness function, aiming to better assess the plausibility of a replacement expression. Specifically, we calculate the difference between the computed value of the modified program and the expected output for each test case, taking the absolute value of this result. The objective is to measure the extent to which the program variant (the replacement expression or patch) deviates from correctness for each test case. By performing the same calculation for all utilized test cases and summing the results, we obtain the fitness function value for the patch. The patch is considered plausible when the value of its fitness function is zero. This method of fitness calculating is different from that in the basic algorithm. Indeed, we do not count the number of test cases that succeed to evaluate the correction of the program to be repaired, but we calculate how much all the test cases deviate from the expected results. Following this improvement, several functions of our algorithm (Alg. 1) need to be modified. The first one is "fitnessFunction", which should calculate the list of fitness values of individuals in order according to the method explained above. The second one is "selectBestIndividuals", which should return the list of the best individuals in *population*. An individual is considered more fit than another if its fitness value is lower (instead of higher), by flipping the fitness function, lower fitness values correspond to better individuals in terms of their proximity to the expected results. Finally, the "existSolution" and "bestIndividuals" functions should, respectively, test if there is one or more individuals with a fitness value equal to zero and, if so, return this list of individuals.

Initial population This improvement is limited to programs that need to be repaired, where assignment and return statement expressions always yield an integer, and all variables are integers. The only Boolean expressions with arithmetic are those within conditional statements. Currently, the process of defining the variables required to construct the expressions (individuals) of the initial population is manual. We aim to automate this process by collecting them from the program to be repaired, specifically from the instruction path leading to the instruction that requires repair. Variables should be either local to the function (or procedure) containing the expression to be repaired or global. Local variables should be modified after declaration. There is no requirement to modify global variables in order to utilize them. The parameters of the function (or procedure) containing the expression to be repaired are also included in the set of variables used to construct the initial population. Let's consider the example of the AbsMinus program[4] (Listing 3). The set of variables to be used for correcting the expression in line 9 is $\{i, j, k\}$. Although *result* is declared along the

[4]The AbsMinus correct program returns the absolute value of i minus j (i and j are the inputs).

path leading to the instruction that requires repair, it is only modified at this instruction or in line 12, which is on a different path. If we had global variables, we could include them in the set of variables without any issue.

Listing 3. The AbsMinus program with an error

```
1   public class AbsMinus {
2           public static int absM (int i, int j) {
3                   int result;
4                   int k = 0;
5                   if (i <= j) {
6                           k = k+1;
7                   }
8                   if (k == 1 && i != j) {
9                           result = i-j; //error: should be
                                result = j-i
10                  }
11                  else {
12                          result = i-j;
13                  }
14                  return result;
15          }
16  }
```

After the variables, the process of defining the set of constants should be automated. However, before explaining how we proceed with that, let us discuss an improvement that could accelerate the search for a plausible patch. This improvement involves adding instruction expressions from the program to be repaired to the initial population, provided they compile successfully[5]. We have two scenarios: either the error is in an assignment statement, in which case we only consider the expressions from assignment statements, or it is in a branching instruction, which means that only the expressions from conditional instructions are added. Applying this principle to the AbsMinus program shown in Listing 3, we would add the expressions "0", "$k+1$", and "$i-j$" to the initial population (the error was in an assignment). If the error was in a branching instruction, we would add the expressions "$(i <= j)$" and "$(k == 1 \&\& i! = j)$". We construct the set of constants from the expressions added to the initial population: we parse each expression and whenever we encounter a constant, we add

[5]This means that if we replace the instruction expression to be repaired with each of these expressions, the program compiles without errors.

it to this set. In the case of the AbsMinus program example, the set of constants would be $\{0, 1\}$ (for an error in an assignment). If the error was in a condition, then the set would be $\{1\}$. The set of operators to be determined depends on the instruction to be repaired: if the error is in an assignment, it will contain the usual arithmetic operators; otherwise, it will consist of arithmetic operators, comparison operators, and Boolean operators.

5 Results

We used Java programming language to develop our approach and to test it in practice. This implementation, called GenExp, is an extension on OakGP, an open-source GP framework written in Java[6]. This initial implementation can only correct integer expressions, and the operators considered are: multiplication ($*$), division ($/$), addition ($+$), subtraction ($-$), logical OR ($||$), logical AND (&&), NOT (!), is equal to ($==$), is not equal to ($! =$), is less than ($<$), is greater than ($>$), is greater than or equal ($>=$), and is less than or equal ($<=$). Our implementation includes the improvements explained in the previous section. We have also conducted a series of experiments on a number of toy programs that we built ourselves, which require a correction in a single location. The considered error can be either in an assignment or in a conditional statement.

Table 2. Programs used the experiments

Programs class	Nbr programs	Nbr tests
AbsMinus	10	9
BSearch	6	9
BubbleSort	7	5
Gcd	7	25
Heron	5	9261
Maxmin6var (M6var)	3	4096
Mid	10	36
Minimum (Min)	2	7
SquareRoot	6	15
Tritype	7	125
Total	63	

[6]OakGP is available at this link : `http://www.oakgp.org/`.

Table 2 shows the programs used in the experiments[7]. To explore the capabilities of our implementation on a program (column "Programs class"), we introduce an error in one of its instruction expressions and run the tool to determine if a correction can be found (i.e., a replacement expression that passes all test cases considered). The number of test cases employed is indicated in the "Nbr tests" column. This process is repeated multiple times by injecting a different error each time (see "Nbr programs" column). All developed repair programs meet the prerequisites described in Subsection 4.1.

We conducted a performance comparison between our implementation and ASTOR [6], an Automatic Software Transformations for Program Repair tool (refer to Subsection 3.2). The modes of this tool utilized in our experiments included JGenProg, MutRepair, and CARDUMEN.

Our tool repairs erroneous programs by focusing on a single instruction expression that is responsible for the error. In other words, we assume that we have achieved perfect error localization. JGenProg, MutRepair, and CARDUMEN repair the input program by initially identifying a set of suspicious instructions through their first phase of error localization. To ensure a fair comparison with ASTOR's approaches, we not only start with the same erroneous program and utilize the same test cases, but also narrow down the set of suspicious instructions generated during the error localization step to include only the injected erroneous instruction. The repair process concludes once a plausible patch is found for all tools. Additionally, there is a timeout of 5 minutes, after which the process may terminate without a solution. Testing a variant of the program to be fixed is limited to a maximum 20 milliseconds to address cases involving infinite loops.

All experiments were conducted using an Intel Core i7-3720QM processor, clocked at 2.6 GHz and equipped with 8 GB of memory. The experiments were performed on a 64-bit Linux operating system.

In our experiments, we measure the running times by running the tools multiple times on the same input. We record the time taken for each run and select the best time obtained as our metric for per-

[7]Experimental programs are available at `https://sites.google.com/prod/esi-sba.dz/error-correction-experiments`.

formance comparison. This approach allows us to capture the most efficient performance achieved by the tools.

The results of our experiments are summarized in Table 3. The ”B” column lists the benchmark programs used, which correspond to the ”Programms class” column in Table 2. The ”Versions” column shows the erroneous versions created for each of these programs. The third, fourth, fifth, and sixth columns indicate whether CARDUMEN, JGenProg, MutRepair, and our implementation were able to find a plausible patch (✓) or not (✗) for each erroneous program considered. If a tool finds a plausible patch, we also display the time elapsed during the repair process. The ”G” column displays the number of generations reached by our GP-based algorithm.

The results[8] showed that our approach successfully finds a plausible patch for almost all cases and outperforms ASTOR’s tools, even when combined. Figure 2 presents a side-by-side bar chart that compares the CARDUMEN, JGenProg, MutRepair, and GenExp tools in terms of the number of times a plausible patch is found, allowing to compare for each benchmark program.

The low number of generations observed in the majority of cases presented in Table 3 can be attributed to several factors that contribute to the efficiency of our approach. One important factor is the inclusion of interesting expressions from the program being corrected in the initial population. These expressions serve as potential building blocks for constructing plausible patches. Additionally, it is important to note that the program itself may already contain an expression that represents the plausible patch. Moreover, the construction of initial sets of variables and constants specifically derived from the program being corrected helps reduce the search space effectively. This reduction enables the genetic algorithm to focus its exploration on the most relevant and promising areas. The fitness function further enhances the convergence process by favoring the selection of superior expressions. As a result, the convergence process is accelerated, leading to the discovery of plausible patches in a shorter number of generations.

Before each new generation, GenExp calculates the fitness value for

[8]Experimental results are also available at `https://sites.google.com/prod/esi-sba.dz/error-correction-experiments`.

Table 3: Summary of the results of our experiments

B	Versions	CARDUMEN	JGenProg	MutRepair	GenExp	G
AbsMinus	1	✓(0,86s)	✗	✗	✓(2,435s)	3
	2	✗	✗	✗	✓(1,435s)	1
	3	✗	✗	✗	✗	\
	4	✓(0,739s)	✗	✗	✓(1,286s)	1
	5	✗	✗	✗	✓(1,531s)	1
	6	✓(0,778s)	✗	✗	✓(2,113s)	2
	7	✓(0,769s)	✗	✗	✓(2,356s)	3
	8	✓(0,751s)	✗	✗	✓(1,747s)	1
	WrongIf1	✓(2,395s)	✓(2,454s)	✓(1,436s)	✓(1,289s)	1
	WrongIf2	✓(0,793s)	✗	✓(0,713)	✓(1,129s)	1
BSearch	1	✗	✗	✗	✓(2,705s)	1
	2	✓(2,657s)	✗	✗	✓(2,267s)	1
	3	✗	✗	✗	✓(5,552s)	2
	WrongIf1	✗	✗	✓(1,166s)	✓(3,7s)	3
	WrongIf2	✗	✗	✓(0,795s)	✗	\
	WrongWhile	✗	✗	✗	✓(129,586s)	70
BubbleSort	1	✗	✗	✗	✓(2,826s)	1
	2	✗	✗	✗	✓(107,059s)	112
	3	✓(0,855s)	✗	✗	✓(2,338s)	2
	4	✓(0,877s)	✓(0,784s)	✗	✓(0,605s)	1
	WrongIf	✓(4,035s)	✗	✓(1,43s)	✓(1,399s)	2
	WrongWhile1	✗	✗	✗	✓(2,084s)	3
	WrongWhile2	✓(9,428s)	✗	✗	✓(1,015s)	1
Gcd	1	✓(1,074s)	✗	✗	✓(33,423s)	6
	2	✓(0,802s)	✗	✗	✓(13,07s)	2
	WrongIf1	✓(0,861s)	✗	✓(2,808s)	✓(3,262s)	1
	WrongI2	✗	✗	✗	✓(2,929s)	1
	WrongIf3	✓(0,912s)	✗	✓(2,729s)	✓(36,293s)	4
	WrongIf4	✓(1,137s)	✗	✗	✓(3,176s)	1
	WrongWhile	✗	✗	✗	✓(8,489s)	1
Heron	1	✗	✓(5,669s)	✗	✓(37,474s)	1
	2	✗	✗	✗	✗	\
	WrongIf1	✓(23,469s)	✗	✓(0,766s)	✓(279,383s)	6
	WrongI2	✗	✗	✗	✗	\
	WrongIf3	✗	✗	✓(7,67s)	✗	\
M6var	WrongIf1	✓(7,33s)	✓(1,15s)	✓(1,577s)	✓(26,138s)	1
	WrongIf2	✗	✗	✗	✗	\
	WrongIf3	✗	✓(7,405s)	✗	✓(257,766s)	12
Mid	1	✗	✓(1,184s)	✗	✓(1,699s)	1
	2	✗	✓(1,151s)	✗	✓(1,16s)	1
	3	✗	✗	✗	✓(0,829s)	1
	4	✗	✓(2,008s)	✗	✓(0,902s)	1
	5	✗	✓(1,191s)	✗	✓(0,805s)	1
	WrongIf1	✓(3,245s)	✗	✓(1,305s)	✓(1,291s)	1
	WrongIf2	✓(2,613s)	✗	✓(1,402s)	✓(3,318s)	3
	WrongIf3	✓(2,766s)	✗	✓(1,23s)	✓(2,347s)	2
	WrongIf4	✓(2,322s)	✗	✓(1,399s)	✓(1,245s)	1
	WrongIf5	✓(1,397s)	✗	✓(1,154s)	✓(1,175s)	1
Min	WrongIf1	✗	✗	✓(1,491s)	✓(2,369s)	1
	WrongWhile	✗	✓(2,466s)	✗	✓(2,068s)	2
SquareRoot	1	✗	✗	✗	✓(2,862s)	5
	2	✗	✗	✗	✓(0,625s)	1
	3	✗	✗	✗	✓(1,349s)	7
	4	✗	✗	✗	✓(2,495s)	4
	5	✗	✗	✗	✓(30,085s)	13
	WrongWhile	✓(1,6s)	✗	✗	✓(4,518s)	1
Tritype	1	✗	✓(0,76s)	✗	✓(1,305s)	1
	WrongIf1	✓(6,452s)	✓(13,711s)	✗	✓(3,569s)	1
	WrongIf2	✗	✓(2,711s)	✗	✓(41,755s)	27
	MultPerimetre	✓(4,121s)	✓(1,378s)	✗	✓(3,13s)	1
	MultPerimetreWrongIf	✓(4,282s)	✓(4,386s)	✗	✓(30,621s)	19
	Perimetre	✓(12,084s)	✓(1,333s)	✗	✓(2,962s)	1
	Perimetre2	✓(3,95s)	✓(3,482s)	✗	✓(2,145s)	1

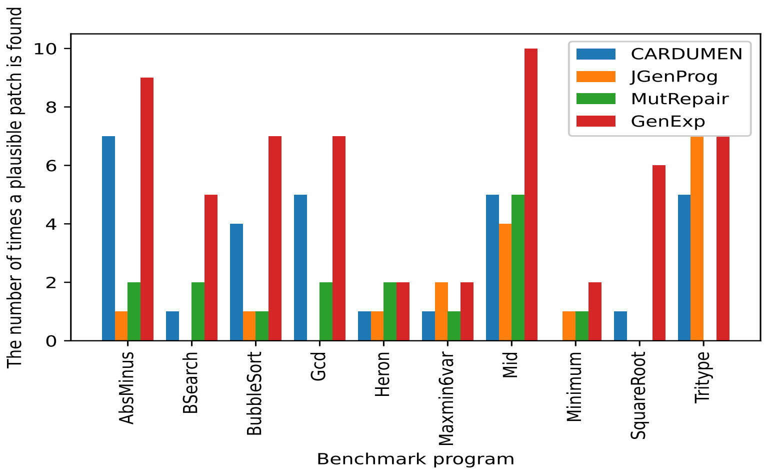

Figure 2. The number of times a plausible patch is found by CARDUMEN, JGenProg, MutRepair, and GenExp for each benchmark program

every replacement expression in the current population. This involves executing each variant of the program under repair, which corresponds to a replacement expression, on all the test cases used as input for that program. It's important to note that the running time in Table 3 does not always correlate directly with the number of generations. For instance, the WrongIf1 version of the Heron program completes 6 generations in 279.383 seconds, while the WrongIf3 version of the Maxmin6var program takes 12 generations in 257.766 seconds. On the other hand, the WrongWhile version of BSearch requires 70 generations in 129.586 seconds, and the version 2 of BubbleSort completes 122 generations in 107.059 seconds. These discrepancies can be attributed to the number of test cases used for each benchmark. The Heron and Maxmin6var benchmarks use 9261 and 4096 test cases, respectively, while the BSearch and BubbleSort benchmarks use only 9 and 5 test cases, respectively (refer to Table 2).

5.1 Limitation

To summarize and analyse the repair times of each tool used in our experiments, we have presented basic statistics such as the mean, median, standard deviation, as well as the minimum and maximum values in Table 4. We only consider the times corresponding to cases where the tools have identified a plausible patch.

Table 4. The basic statistics on the execution times obtained in Table 3

	CARDUMEN	JGenProg	MutRepair	GenExp
Mean	$3,602s$	$3,131s$	$1,818s$	$19,728s$
Median	$2,359s$	$2,008s$	$1,401s$	$2,369s$
Standard deviation	$4,656$	$3,293s$	$1,669s$	$53,313s$
Minimum	$0,739s$	$0,76s$	$0,731s$	$0,605s$
Maximum	$23,469s$	$13,711s$	$7,67s$	$279,383s$

The results suggest that GenExp takes much longer on average than the other tools to produce repair results, with an average of 19.7 seconds and a large variation in time, ranging from 0.6 to 279.4 seconds. However, it is interesting to note that the median time of GenExp is similar to that of the other tools, indicating that it most often finds a plausible repair within a reasonable time.

Tools CARDUMEN, JGenProg, and MutRepair have much faster average times to produce repair results, with averages ranging from 1.8 to 3.6 seconds and minimum times under 1 second. These tools may be more convenient for users who need quick results, but their accuracy may be lower than that of GenExp.

Figure 3 depicts a line plot with each tool represented by a line. The various statistics (mean, median, standard deviation, minimum, and maximum) are plotted based on the tool. This enables the visualization of the evolution of execution times for each tool.

The significant difference in the range of running times, where the range of GenExp is approximately 10 times greater than that of CARDUMEN and about 30 times greater than that of MutRepair, can be attributed to two factors.

Firstly, the success rate of GenExp is much higher than that of other tools, meaning that there are many more values (times) to consider

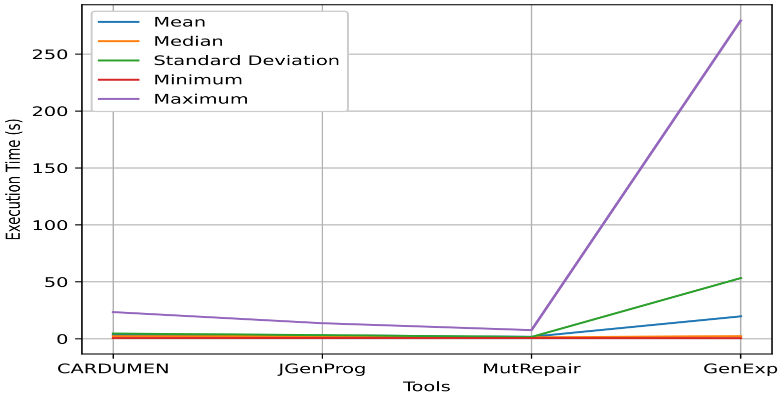

Figure 3. Evolution of execution times by tool

for calculating the standard deviation and other statistics. Specifically, there are 57 values for GenExp, 30 for CARDUMEN, 17 for JGenProg, and 16 for MutRepair. Assuming that GenExp failed to repair the following versions: BSearch (WrongWhile), BubbleSort (version 2), Gcd (version 1 and WrongIf3), Heron (WrongIf1), Maxmin6var (WrongIf3), SquareRoot (version 5), and Tritype (WrongIf2). Consequently, the standard deviation for GenExp decreases from 53,313s to 7,392s, and the success rate is now 49 out of 63. It should be noted that the success rates are 30/63 for CARDUMEN, 17/63 for JGenProg, and 16/63 for MutRepair.

Secondly, the difference can also be attributed to the search space utilized by each tool. JGenProg's search space is limited to the program instructions to be repaired and cannot correct the error using code that is not part of the program. The MutRepair algorithm focuses on mutating the operators within ”if” conditions (see Subsection 3.2). CARDUMEN, as a template-based approach, has a wide search space but not as extensive as our approach since it is limited to templates extracted from the code to be repaired. GenExp has a broader search space compared to the ASTOR tools. While we do utilize the code under repair to construct the sets of variables, operators, and constants necessary for building replacement expressions, we also leverage the repair program to enhance the initial population with expressions extracted from it. However, our algorithm is capable of generating new expressions that cannot be derived from the code being repaired. This explains the high success rate achieved by our tool, which in turn accounts for the difference in the range of running times observed in all experiments.

The large variation in GenExp's times may be problematic for users who need consistent, fast results. It may be preferable to use GenExp when increased accuracy is needed, even if it takes longer.

Although GenExp has the capability to repair the expression of a single instruction, it should be noted that programs containing two or more different erroneous instructions cannot be repaired using this tool. This limitation is also present in CARDUMEN, JGenProg, and MutRepair, as they too are designed to repair input programs from a single suspect instruction. Another constraint of GenExp is its inability to repair programs that require the addition or removal of instructions. Similar limitations exist in CARDUMEN and MutRepair, but JGenProg has the potential to address such errors. Additionally, JGenProg can also correct the left-hand side of an assignment, a capability not available in our tool, CARDUMEN, or MutRepair. Another notable limitation of GenExp is its inability to generate expressions where numbers and variables are not integers. In contrast, CARDUMEN, JGenProg, and MutRepair have the capability to handle variables or constants of any kind.

6 Conclusion

To correct a program, our approach is based on using GP to construct a new program without errors. We utilize the suspected instructions generated by an error locating tool. The error correction process is applied to a selected instruction, chosen by the programmer, which is potentially the source of the problem in the input program. Genetic operators are employed to evolve the expression of this instruction, aiming to generate an individual that produces the expected outputs for the given test cases. The results demonstrate the effectiveness of our approach in program correction, as our implementation successfully fixed nearly all cases used in the experiments. GenExp has outperformed ASTOR's three approaches (JGenProg, MutRepair, and CARDUMEN) in finding plausible fixes for the set of academic repair programs we have developed.

Based on the results, it seems that GenExp is a slower option compared to the other tools used in the experiments, but it is also more accurate in finding plausible repairs. This makes it a good choice for users who prioritize accuracy over speed. However, the large variation in time and the longer average time may be a concern for users who need consistent, fast results. Therefore, it is important to consider the specific needs of the user when deciding whether to use our tool or one of the faster tools.

In most cases, the programmer corrects only one instruction in the set obtained during the error locating step. However, a challenge arises when the program requires corrections in multiple instructions. Our current approach is unable to find a plausible patch in such scenarios since it focuses on a single instruction. As part of future work, we aim to address this limitation by extending our approach to handle multiple suspected instructions. One possible solution could involve running the GP process evolution on multiple suspected instructions.

Currently, GenExp possesses the ability to generate instruction expressions, encompassing both algebraic and boolean expressions with arithmetic operations. Moreover, GenExp generates integer-based expressions where numbers and variables are restricted to the integer domain. Our benchmark is carefully designed to showcase GenExp's capabilities, emphasizing that the correct expressions obtained should exclusively involve integer values for numbers and variables. Moving forward, our objective is to enhance our implementation to accommodate various expression types, enabling comprehensive testing and comparison with ASTOR tools across a wider range of expression classes.

7 Funding

No funding was received to assist with the preparation of this manuscript.

References

[1] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Automatic program repair," *IEEE Software*, vol. 38, no. 4, pp. 22–27, 2021.

[2] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 2008, pp. 162–168.

[3] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[4] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 65–86.

[5] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[6] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[7] Q. Zhang, Y. Zhao, W. Sun, C. Fang, Z. Wang, and L. Zhang, "Program repair: Automated vs. manual," *arXiv preprint arXiv:2203.05166*, 2022.

[8] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[9] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[10] A. Arcuri, “Evolutionary repair of faulty software,” *Applied soft computing*, vol. 11, no. 4, pp. 3494–3514, 2011.

[11] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[12] T. Ji, L. Chen, X. Mao, and X. Yi, “Automated program repair by using similar code containing fix ingredients,” in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 197–202.

[13] Y. Yuan and W. Banzhaf, “Arja: Automated repair of java programs via multi-objective genetic programming,” *IEEE Transactions on software engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[14] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “Tbar: Revisiting template-based automated program repair,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[15] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *2013 35th International Conference on Software Engineering (ICSE).* IEEE, 2013, pp. 802–811.

[16] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, “Dynamic patch generation for null pointer exceptions using metaprogramming,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2017, pp. 349–358.

[17] F. Long, P. Amidon, and M. Rinard, “Automatic inference of code transforms for patch generation,” in *Proceedings of the 2017 11th*

Joint Meeting on Foundations of Software Engineering, 2017, pp. 727–739.

[18] X. Liu and H. Zhong, “Mining stackoverflow for program repair,” in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018, pp. 118–129.

[19] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[20] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, “Repairing programs with semantic code search (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 295–306.

[21] K. T. Stolee, S. Elbaum, and D. Dobos, “Solving the search for source code,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 3, pp. 1–45, 2014.

[22] S. Mechtaev, J. Yi, and A. Roychoudhury, “Directfix: Looking for simple program repairs,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 448–458.

[23] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[24] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, “Semantic program repair using a reference implementation,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 129–139.

[25] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[26] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 832–837.

[27] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “Coconut: combining context-aware neural translation models using ensemble for program repair,” in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[28] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[29] Y. Li, S. Wang, and T. N. Nguyen, “Dear: A novel deep learning-based approach for automated program repair,” *arXiv preprint arXiv:2205.01859*, 2022.

[30] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “An evaluation of similarity coefficients for software fault localization,” in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC’06)*. IEEE, 2006, pp. 39–46.

[31] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[32] V. Debroy and W. E. Wong, “Using mutation to automatically suggest fixes for faulty programs,” in *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 65–74.

[33] M. Bekkouche, H. Collavizza, and M. Rueher, “Locfaults: A new flow-driven and constraint-based error localization approach,” in

Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015, pp. 1773–1780.

[34] M. Bekkouche, "Combining techniques of bounded model checking and constraint programming to aid for error localization," *Constraints*, vol. 22, no. 1, pp. 93–94, 2017.

[35] M. Jose and R. Majumdar, "Bug-assist: assisting fault localization in ansi-c programs," in *International conference on computer aided verification.* Springer, 2011, pp. 504–509.

[36] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 437–446, 2011.

[37] "OakGP Genetic Programming Framework," http://www.oakgp.org/index.html. [Online]. Available: http://www.oakgp.org/index.html

Mohammed Bekkouche

Received April 02, 2023
Revised 1 – June 16, 2023
Revised 2 – July 11, 2023
Accepted July 13, 2023

Mohammed Bekkouche
ORCID: https://orcid.org/0000-0002-8305-0542
LabRI-SBA Laboratory, Ecole Superieure en Informatique,
Sidi Bel Abbes, Algeria
BP. 73, Bureau de poste EL WIAM, Sidi Bel Abbes, 22016, Algeria
E–mail: m.bekkouche@esi-sba.dz