METARO^3: Metamorphic Relation Group for Automatic Program Repair

Tingting Wu[1] and Yunwei Dong[1,*]

Abstract: The application of metamorphic testing (MT) on automatic program repair (APR-MT) is used to generate a patch without test oracles by examining whether the input metamorphic relation (MR) is satisfied or not. However, the delivered patch is plausible since it may satisfy the input MR but violate other MRs. This inspires us to propose an improved approach to enhance the effectiveness of APR-MT with metamorphic relation group. Our approach involves three major steps. First, we formally define the repair process of APR-MT by building the model of automatic program repair and metamorphic testing separately. Then, we propose the advanced model of automatic program repair based on metamorphic relation group, named METARO^3, which takes several MRs as input while only one MR is used in APR-MT. We additionally present two kinds of selection strategies to rank MRs in descending order of the fault detection capability, which helps shorten the repair time of finding a patch. To demonstrate the feasibility and procedure of our approach, an illustration example was conducted. The results show that METARO^3 can improve the effectiveness of APR-MT significantly.

Keywords: Automatic program repair, metamorphic testing, metamorphic relation, formal modeling.

1 Introduction

Program repair aims at fixing the errors in program revealed during the debugging procedure so that the revised version can pass the entire input test cases and no new mistakes are introduced to the revised version during the repair procedure. Since the manual debugging and repair processes are extremely time-consuming and labor-intensive activities in software development, automatic program repair (APR) [Kong, Zhang, Wong et al. (2018); Liu, Zhang and Zhang (2018); Gazzola, Micucci and Mariani (2017)] is proposed to automate both processes.

Many different test suite based APR techniques have been proposed. They fall roughly into two main categories, namely the Generate-and-Validate (GaV) techniques and Correct-

[1]School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an, 710072, China.
[*]Corresponding Author: Yunwei Dong. Email: yunweidong@nwpu.edu.cn.

by-Construction (CbC) techniques [Le Goues, Holtschulte, Smith et al. (2015)]. The GaV techniques, search based techniques, generate multiple revised versions, referred to as candidate repairs, and then validate them with selected test cases (e.g., GenProg [Le Goues, Dewey Vogt, Forrest et al. (2012); Le Goues, Nguyen, Forrest et al. (2012)], AE [Weimer, Fry and Forrest (2013)], TrpAutoRepair [Qi, Mao and Lei (2013)], SPR [Long and Rinard (2015)] and Astor [Martinez and Monperrus (2016, 2019)]). On the other hand, the CbC techniques, semantics based techniques, fix programs by program synthesis or constraint solving to generate patches (e.g., CETI [Nguyen (2014)], SemFix [Nguyen, Qi, Roychoudhury et al. (2013)], Angelix [Mechtaev, Yi and Roychoudhury (2016)], Tortoise [Weiss, Guha and Brun (2017)]).

The generic repair procedure takes as input one buggy program and a test suite including at least one failed test case. It applies the technique of fault localization [Tu, Xie, Chen et al. (2019)] to locate the suspicious program entities. And then it ranks these entities in descending order of their risk values calculated with the risk evaluation formulas. The larger risk value is, the more likely faulty the entity is. Debuggers then inspect the program entities from top to bottom and repair these faults with some program repair techniques. If any candidate repair passes the input test suite, it will be returned as the program repair; otherwise, no repair is found. However, the generated repair is plausible, namely possibly not "correct", since the input test suite is only a partial subset of the entire input domain. Hence, an independent test suite, denoted as evaluation test suite [Wu, Dong, Chen et al. (2017)], is used to assess the repair quality of the generated patch. The more evaluation test cases the patch passes, the better the quality is.

The traditional test suite based APR delivers a program repair with the assumption of the existence of test oracles [Jiang, Chen, Kuo et al. (2017b)] to determine whether a candidate repair can pass the input and evaluation test suite. It is inevitable to have the oracle problem [Barr, Harman, McMinn et al. (2015)] since there are some programs that do not have test oracles or are too expensive to compute oracles. Metamorphic testing (MT) [Chen, Cheung and Yiu (1998); Chen, Kuo, Liu et al. (2018)] is an approach proposed to alleviate the test oracle problem by checking whether the metamorphic relation (MR), the relation between multiple input test cases and outputs, is satisfied. To solve the oracle problem in program repair, Jiang et al. [Jiang, Chen, Kuo et al. (2017b)] propose an integration approach of metamorphic testing and automatic program repair (APR-MT). This framework takes as input a set of metamorphic testing groups (MTG) formed by one MR, source test cases and follow-up test cases instead the test suite in APR. APR-MT determines the correctness of candidate patch by checking whether the input MTG is satisfied or not. If the entire MTG is satisfied, that is the related MR is satisfied, then this candidate patch is the program repair debuggers desired. Once any metamorphic testing group (mtg) is violated, no repair is found.

However, the patch generated from APR-MT is probably plausible, because it may satisfy the input MR but violate other MRs. In other words, the generated patch is not correct and the APR-MT is of low repair quality which affects the repair effectiveness fundamentally. This leads us to propose an advanced repair approach based on metamorphic relation group, aiming to answer the following research question:

Will multiple input MRs improve the effectiveness of APR-MT significantly?

We first define the model of APR-MT to precisely describe the repair procedure that how to generate a patch without oracles. Based on which, we apply multiple MRs as input to present a model of automatic program repair based on metamorphic relation group, METARO^3. If a patch is found to satisfy all of the input MRs, it will be returned as the program repair; otherwise, no repair is generated. Additionally, since the fault detection capability varies with different MRs, we also study the selection strategy to rank the input MRs with the order of their detection capability which can help debuggers to examine the most powerful MR first and find and fix errors faster.

The reminder of this paper is organized as follows. Section 2 presents the preliminaries of automatic program repair, including test suite based APR and metamorphic testing based APR. In Section 3, we define the model of test suite based APR, MT and APR-MT, based on which we define the model and algorithm of METARO^3. Section 4 introduces two selection strategies for metamorphic relations according to their fault detection capability. Section 5 discusses our illustration example with the “mid” program. Section 6 discusses the threats to validity of our approach. Section 7 briefly reviews the most recent work in program repair. Section 8 concludes the paper and discusses future work.

2 Preliminaries

2.1 Test suite based automatic program repair

There are mainly two categories of test suite based APR: Generate-and-Validate and Correct-by-Construction techniques [Jiang, Chen, Kuo et al. (2017b); Le Goues, Holtschulte, Smith et al. (2015)]. The GaV techniques [Long and Rinard (2016a)] generate multiple candidate patches and validate them with the input test suite to deliver one patch passing the entire input test suite. GenProg [Le Goues, Nguyen, Forrest et al. (2012)] is a typical GaV technique based on genetic algorithm. It first initializes a set of candidate patches, and then selects a candidate patch with higher fitness through the fitness function. GenProg stops if a candidate patch is found to satisfy the entire input test suite; otherwise no repair is found within the maximum number of generations. Prophet [Long and Rinard (2016b)] is a repair technique based on the probabilistic model of correct code which ranks the candidate patches with their correctness. Opad [Yang, Zhikhartsev, Liu et al. (2017)] is a framework to detect overfitted patch by generating better test cases using fuzz testing. Elixir [Saha, Lyu, Yoshida et al. (2017)] is designed for repairing the object-oriented programs by constructing expressive expressions using method invocations and ranked with machine-learnt model. SOFix [Liu and Zhong (2018)] fixes bugs with 13 patch patterns by applying code fragments extracted from stack overflow. LSRepair [Liu, Koyuncu, Kim et al. (2018)] aims at finding fixed ingredients with three search strategies of similar code at the method level.

The CbC techniques generate program repair based on semantics which replaces faulty expressions with constraint conditions encoding the characterized information of test suite. SemFix [Nguyen, Qi, Roychoudhury et al. (2013); Roychoudhury (2016)] is the first program repair technique applied semantics. MintHint [Kaleeswaran, Tulsian, Kanade et al. (2014)] utilizes the symbol execution to replace and instantiate faulty expression and executes patch successfully. Nopol [Xuan, Martinez, Demarco et al. (2017)] delivers a repair by fixing buggy conditional statements which are executed by failed test cases. DeepFix [Gupta, Pal, Kanade et al. (2017)] is an approach based on deep learning which targets at compilation errors in C programs. SemGraft [Mechtaev, Nguyen, Noller et al. (2018)] solves the test overfitting problem with symbolic analysis of behavior specification for buggy program from a correct reference implementation. ACS [Xiong, Wang, Yan et al. (2017)] synthesizes precise condition by both variable ranking and predicate ranking.

2.2 Metamorphic testing based automatic program repair

The APR-MT [Jiang, Chen, Kuo et al. (2017b)] technique integrates metamorphic testing with program repair, aiming at solving the oracle problem in program repair and generating a repair by inspecting the input MR. APR-MT has been currently applied on three APR techniques which demonstrates the feasibility of the integrated technique. GenProg-MT [Jiang, Chen, Kuo et al. (2017b)] utilizes the principle of GenProg to generate the set of candidate patches with mutate operators, and then investigates the candidate patch according to the fitness calculated from the input MTG and fitness function. The experimental results show that the effectiveness of GenProg-MT is comparable to GenProg. AE-MT [Wu, Dong, Chen et al. (2017)] prioritizes the candidate patches to choose one that is most likely to satisfy the entire MTG according to the adaptive repair strategy. AE-MT then evaluates the chosen patch with the input MTG, and it favors the mtg that is most likely to be violated as early as possible according to the adaptive test strategy. AE-MT shows a great advantage of repair time compared with GenProg-MT. Both GenProg-MT and AE-MT are the applications of APR-MT on GaV techniques. CETI-MT [Jiang, Chen, Kuo et al. (2017a)] applies the APR-MT on a CbC technique. CETI-MT constructs a reachability instance program from a faulty program by replacing a suspicious statement with a parameterized statement encoding all the requirements of MTG set. CETI-MT then checks the reachability by an independent MR checking function. Once the relevant MR on the input MTG set is satisfied, this instance program is considered reachable and a repair for the given faulty program is found as well.

3 Application of metamorphic relation group on automatic program repair

3.1 Model of test suite based automatic program repair

Test suite based APR takes as input faulty program and test suite with at least one failed test case and evaluates the correctness of patch by comparing the actual outputs and expected outputs. If the candidate patch passes the entire test suite, it will be regarded as the program repair to generate. According to the repair procedure, we define the repair and evalution processes by a six tuple as follows. This model formally depicts the inputs of APR, the buggy program p_0 and test suite T. The repair and evalution processes are defined by function f and g as well, which represent the semantic of program repair precisely.

Definition 3.1. The model of test suite based APR is defined as a tuple, $APR =$

$(P, T, \Sigma, f, g, p_0)$, in which

(1) P is a set of programs including the faulty program and repaired versions, $P = \{p_0, p^1_{patch}, p^2_{patch}, \cdots, p^j_{patch}, \cdots, p^k_{patch}\}$. Among which, p_0 is the faulty program to be fixed and p^j_{patch} is the repaired version, candidate patch, generated from the j^{th} repair operation;

(2) T is a set of test cases, $T = \{t_1, t_2, \cdots, t_i, \cdots, t_n\}$;

(3) Σ is a set of repair operations on p_0, $\Sigma = \{r^1_{repair}, r^2_{repair}, \cdots, r^j_{repair}, \cdots, r^k_{repair}\}$. r^j_{repair} indicates the j^{th} repair activity on buggy program with the corresponding candidate patch p^j_{patch} generated;

(4) $f : \{p_0\} \times \Sigma \rightarrow \{p^1_{patch}, p^2_{patch}, \cdots, p^j_{patch}, \cdots, p^k_{patch}\}$ is a function from buggy program and repair operations to candidate patches. $f(p_0, r^j_{repair}) = p^j_{repair}$ implies that candidate patch p^j_{repair} is generated from the j^{th} repair activity r^j_{repair} on buggy program;

(5) $g : (P - \{p_0\}) \times T \rightarrow \{0, 1\}$ is a decision function of execution result for test suite T examined on candidate patch p^j_{patch}, in which 1 indicates test case p^j_{patch} passes t_i and 0 indicates p^j_{patch} fails t_i. If p^j_{patch} passes the entire suite T, p^j_{patch} is regarded as the final patch to output; otherwise, investigate the next candidate patch or no repair generated. That is, $\forall t_i \in T, p^j_{patch} \in P - \{p_0\}$, $\exists p^j_{patch}$ to let $g(p^j_{patch}, t_i) = 1$, then p^j_{patch} will be the final generated patch for buggy program; $\forall t_i \in T, p^j_{patch} \in P - \{p_0\}$, $\exists t_i$ to let $g(p^j_{patch}, t_i) = 0$, then terminate inspecting p^j_{patch} and turn to p^{j+1}_{patch}; If $j = k$ and $\exists t_i$ to let $g(p^k_{patch}, t_i) = 0$, then no repair is generated by program repair technique.

3.2 Model of metamorphic testing

The model of metamorphic testing is defined by describing the program under testing p, source test suite T_s, follow-up test suite T_f and the source and follow-up output sets O_s and O_f, which aim at defining the relation r and r_f. The formal description of MR is then defined according to r and r_f subsequently. Therefore, metamorphic testing is formally defined by the above elements.

Definition 3.2. Metamorphic testing is defined as a set, $MT = \{p, T_s, O_s, T_f, O_f, r, r_f, MR\}$, in which

(1) $p \in P$ is a program version from program set P executed currently;

(2) $T_s \subseteq T$ is a set of source test cases, $T_s = \{t^1_s, t^2_s, \cdots, t^m_s\}$ and $m \leqslant n$;

(3) O_s is the set of source outputs of program p executed on T_s, $O_s = \{o^1_s, o^2_s, \cdots, o^m_s\}$. Namely, $\forall t^i_s \in T_s, \exists o^i_s$ to have $o^i_s = p(t^i_s)$;

(4) T_f is the set of follow-up test cases, $T_f = \{t_f^1, t_f^2, \cdots, t_f^m\}$;

(5) O_f is the set of follow-up outputs of program p executed on T_f, $O_f = \{o_f^1, o_f^2, \cdots, o_f^m\}$. Namely, $\forall t_f^i \in T_f, \exists o_f^i$ to have $o_f^i = p(t_f^i)$;

(6) r is the relation between T_s and T_f, $r = \{\left\langle t_s^1, t_f^1 \right\rangle, \left\langle t_s^2, t_f^2 \right\rangle, \cdots, \left\langle t_s^m, t_f^m \right\rangle\}$. That is, $\forall t_s^i \in T_s, \exists t_f^i \in T_f, t_f^i = r(t_s^i)$;

(7) r_f is the relation between O_s and O_f, $r_f = \{\left\langle o_s^1, o_f^1 \right\rangle, \left\langle o_s^2, o_f^2 \right\rangle, \cdots, \left\langle o_s^m, o_f^m \right\rangle\}$. That is, $\forall o_s^i \in O_s, \exists o_f^i \in O_f, o_f^i = r_f(o_s^i)$;

(8) MR is the property that program p should satisfy. If T_s and T_f satisfy relation r, O_s and O_f will satisfy relation r_f. That is, $r(T_s, T_f) \Rightarrow r_f(O_s, O_f)$, then (r, r_f) is one of the MRs of program p.

3.3 Model of APR-MT

Since the integration of MT and APR alleviates the oracle problem in program repair, we define the APR-MT by redefining the input test suite and evaluation process. APR-MT replaces the input test suite T with the set of metamorphic testing groups $MTGs$. If the candidate patch satisfies all of the metamorphic testing groups, that is the input MR is satisfied, then this patch will be output as the program repair; otherwise, no repair is found. The model of APR-MT is defined as follows.

Definition 3.3. The model of program repair based on metamorphic testing is defined as a tuple as well, $APR - MT = \{P, MTGs, \Sigma, f, g, p_0\}$, in which

(1) P is a set of programs including the faulty program and repaired versions, $P = \{p_0, p_{patch}^1, p_{patch}^2, \cdots, p_{patch}^j, \cdots, p_{patch}^k\}$. Among which, p_0 is the faulty program to be fixed and p_{patch}^j is the repaired version, candidate patch, generated from the jth repair operation;

(2) $MTGs$ is the set of metamorphic testing groups formed by source test suite T_s, follow-up test suite T_f and corresponding metamorphic relation MR, $MTGs = \{mtg_1, mtg_2, \cdots, mtg_m\}$ in which $mtg_i = (t_s^i, t_f^i)$;

(3) Σ is a set of repair operations on p_0, $\Sigma = \{r_{repair}^1, r_{repair}^2, \cdots, r_{repair}^j, \cdots, r_{repair}^k\}$. r_{repair}^j indicates the jth repair activity on buggy program with the corresponding candidate patch p_{patch}^j generated;

(4) $f : \{p_0\} \times \Sigma \rightarrow \{p_{patch}^1, p_{patch}^2, \cdots, p_{patch}^j, \cdots, p_{patch}^k\}$ is a function from buggy program and repair operations to candidate patches. $f(p_0, r_{repair}^j) = p_{repair}^j$ implies that candidate patch p_{repair}^j is generated from the jth repair activity r_{repair}^j on buggy program;

(5) $g : (P - \{p_0\}) \times MTGs \to \{0, 1\}$ is a decision function of execution result of $MTGs$ examined on candidate patch p^j_{patch}, in which 1 indicates metamorphic testing group mtg_i is satisfied by p^j_{patch} and 0 indicates mtg_i is violated by p^j_{patch}. If p^j_{patch} satisfies all of the $mtgs$ in set $MTGs$, that is, p^j_{patch} satisfies the input MR, then p^j_{patch} is regarded as the final patch to output; otherwise, investigate the next candidate patch or no repair is delivered. That is, $\forall mtg_i \in MTGs, p^j_{patch} \in P - \{p_0\}$, $\exists p^j_{patch}$ to let $g(p^j_{patch}, mtg_i) = 1$, then p^j_{patch} will be the generated patch for buggy program; or $\forall mtg_i \in MTGs, p^j_{patch} \in P - \{p_0\}$, $\exists mtg_i$ to let $g(p^j_{patch}, mtg_i) = 0$, then terminate inspecting p^j_{patch} and turn to p^{j+1}_{patch}; If $j = k$ and $\exists mtg_i$ to let $g(p^k_{patch}, mtg_i) = 0$, then no repair is generated by program repair technique.

3.4 Automatic program repair based on metamorphic relation group

3.4.1 Model of METARO^3

The patch generated from APR-MT is possibly plausible. That is, this patch may satisfy the input metamorphic relation but violate other metamorphic relations which will weaken the effectiveness of APR-MT, especially the repair quality. To improve the repair effectiveness of APR-MT, we propose to use the metamorphic relation group including several metamorphic relations, at least two MRs. Thus, applying MRG on APR involves three key steps: (1) taking several MRs and corresponding sets of metamorphic testing groups as input instead of one MR in APR-MT; (2) the generation of candidate patch depends on several MRs instead of one MR during repair process; and (3) only if all of input MRs are satisfied, the buggy program is then said to be repaired and output this candidate patch; otherwise, there is no patch generated. The model of METARO^3 is defined as follows.

Definition 3.4. The model of program repair based on metamorphic relation group is defined as a tuple similarly, METARO^3$= \{P, MRG, \Sigma, f, h, p_0\}$, in which

(1) P is a set of programs including the faulty program and repaired versions, $P = \{p_0, p^1_{patch}, p^2_{patch}, \cdots, p^j_{patch}, \cdots, p^k_{patch}\}$. Among which, p_0 is the faulty program to be fixed and p^j_{patch} is the repaired version, candidate patch, generated from the j^{th} repair activity;

(2) MRG is grouped of n metamorphic relations and n related sets of metamorphic testing groups, $MRG = \{MTG_1, MTG_2, \cdots, MTG_i, \cdots, MTG_n\}$. In which, MTG_i is the set of metamorphic testing groups formed by MR_i and related source and follow-up test cases sized m, and $MTG_i = \{mtg^1_i, mtg^2_i, \cdots, mtg^j_i, \cdots, mtg^m_i\}$. MTG_i is regarded as a positive test case if it is satisfied by program; otherwise MTG_i is a negative test case if violated;

(3) Σ is a set of repair operations on p_0, $\Sigma = \{r^1_{repair}, r^2_{repair}, \cdots, r^j_{repair}, \cdots, r^k_{repair}\}$. r^j_{repair} indicates the j^{th} repair activity on buggy program with the corresponding

candidate patch p^j_{patch} generated;

(4) $f : \{p_0\} \times \Sigma \rightarrow \{p^1_{patch}, p^2_{patch}, \cdots, p^j_{patch}, \cdots, p^k_{patch}\}$ is a repair function for buggy program. $f(p_0, r^j_{repair}) = p^j_{repair}$ implies that p_0 is fixed to generate candidate patch p^j_{repair} from the j^{th} repair activity r^j_{repair};

(5) $h : (P - \{p_0\}) \times MRG \rightarrow \{0, 1\}$ is a decision function of execution result of MRG examined on candidate patch p^j_{patch}, in which 1 indicates the set MTG_i is satisfied by p^j_{patch} and 0 indicates MTG_i is violated by p^j_{patch}. In other words, MR_i is satisfied or violated by p^j_{patch}. If p^j_{patch} satisfies all of the MTG_i in MRG, that is, p^j_{patch} satisfies all of the input MRs, then p^j_{patch} is regarded as the final patch to output; otherwise, investigate next candidate patch or no repair is generated. That is, $h(p^j_{patch}, MTG_i) = \wedge^m_{l=1} g(p^j_{patch}, mtg^l_i)$. Only if all of mtg^l_i are satisfied, then all of MR_i and MTG_i are satisfied and $h(p^j_{patch}, MTG_i) = 1$; or $h(p^j_{patch}, MTG_i) = 0$. Hence, $\forall MTG_i \in MRG, p^j_{patch} \in P - \{p_0\}, \exists p^j_{patch}$ to have $h(p^j_{patch}, MTG_i) = 1$, then p^j_{patch} will be the output patch; otherwise, $\forall MTG_i \in MRG, p^j_{patch} \in P - \{p_0\}, \exists MTG_i$ to have $h(p^j_{patch}, MTG_i) = 0$, then terminate inspecting p^j_{patch} and turn to p^{j+1}_{patch}; If $j = k$ and $\exists MTG_i$ to have $h(p^k_{patch}, MTG_i) = 0$, then no repair is generated by program repair technique.

Note that, the input and decision function of METARO^3 differ from that of APR-MT. METARO^3 takes as input MRG, a group of metamorphic relations, while APR-MT takes only one MR as input. Furthermore, the output of function h relies on the output of function g. Only when all of outputs of g are 1, the output of h will be 1 and the buggy program is repaired with the patch generated. Once g outputs one 0, namely one metamorphic testing group is violated, the output of h will be 0 which denotes that inspect next candidate patch or repair terminates.

3.4.2 Algorithm

As defined above, only if all of the metamorphic testing groups for each MR are satisfied, the candidate patch will be regarded as the output patch for the buggy program. Once any metamorphic testing group is violated by the candidate patch, the inspected candidate patch will be discarded and turn to inspect the next candidate patch. When all of the candidate patches are inspected to find that no candidate patch satisfies all of the input MRs, this indicates that no patch is generated by METARO^3. Algorithm (3.1) presents the implementation process of METARO^3 when given a buggy program and a group of metamorphic relations with at least one voilated, and outputs the generated patch or indicates that no patch is generated.

METARO^3 first generates a set of candidate patches CR with the repair operator, some specific repair technique. As defined in **Definition** 3.4, $CR = \{p^1_{patch}, p^2_{patch}, \cdots, p^j_{patch}, \cdots,$

Algorithm 3.1 Automatic Program Repair Based on Metamorphic Relation Group

```
Input: faulty program $p_0$
Input: metamorphic relation group $MRG$ including at least one violated $MR$
Output: program patch $p^{'}$ satisfying all of the input $MRs$ or “no repair”
 1: $CR \leftarrow \emptyset$ // $CR$ is a set of candidate repairs
 2: $\Sigma = \{r^1_{repair}, r^2_{repair}, \cdots, r^j_{repair}, \cdots, r^k_{repair}\}$ // $\Sigma$ is the set of mutation operators
 3: $CR \leftarrow f(p_0, \Sigma)$ // Initialize $CR$ by applying mutation operators on faulty program $p_0$
 4: while $CR \neq \emptyset$ do
 5:    $p^{'} \leftarrow getFrom(CR)$ // Get one candidate repair in $CR$ from top to bottom
 6:    $CR \leftarrow CR \setminus \{p^{'}\}$
 7:    $MRG^{'} \leftarrow selectStrategy(p^{'}, MRG)$ // Get the ordered metamorphic relation
       group $MRG^{'}$ following the selection strategy
 8:    $re \leftarrow 1$ // Initialize the execution result
 9:    while $MRG^{'} \neq \emptyset$ and $re = 1$ do
10:       $MTG \leftarrow getFrom(MRG^{'})$ // Get a set of metamorphic testing groups involving
          one MR in $MRG^{'}$ from top to bottom
11:       $MRG^{'} \leftarrow MRG^{'} \setminus \{MTG\}$
12:       while $MTG \neq \emptyset$ and $re = 1$ do
13:          $mtg \leftarrow getFrom(MTG)$ // Get one metamorphic testing group $mtg$ from
             $MTG$
14:          $MTG \leftarrow MTG \setminus \{mtg\}$
15:          $re \leftarrow g(p^{'}, mtg)$ // Execute $mtg$ on candidate patch $p^{'}$. If $re$ is 0, then terminate
             inspecting candidate patch $p^{'}$
16:       end while
17:    end while
18:    if $MRG^{'} = \emptyset$ and $re = 1$ then
19:       return $p^{'}$
20:    end if
21: end while
22: if $CR \neq \emptyset$ and $re = 0$ then
23:    return “no repair”
24: end if
```

$p^k_{patch}\}$. METARO^3 repeatedly inspects candidate patch $p^{'}$ and ranks the input group of MRs until CR is empty (line 4-line 21), which implements the decision function h in model of METARO^3. Then METARO^3 selects one MR and its relevant metamorphic testing group MTG from the ordered $MRG^{'}$ and executes each mtg on $p^{'}$ (line 9-line 17). If $p^{'}$ satisfies all of the input MRs, $p^{'}$ is regarded as a repair of the buggy program p_0. If each candidate patch $p^{'}$ violates any mtg in the relevant MTG, no repair is generated for the buggy program.

4 Selection strategy of metamorphic relations

The input group of metamorphic relations should be ordered since the detection capability varies with metamorphic relations. Therefore, we present two selection strategies to help debuggers select the most effective metamorphic relations preferentially according to their fault detection capability.

4.1 Negative test case driven

Intuitively, when a large amount of metamorphic testing groups are violated, we deem the relevant MR to be powerful in fault detection and worthwhile to be examined first. For example, given two MRs, MR_1 and MR_2, and two sets of metamorphic testing groups $MTG_1 = \{mtg^1_1, mtg^2_1, \cdots, mtg^m_1\}$ and $MTG_2 = \{mtg^1_2, mtg^2_2, \cdots, mtg^n_2\}$. Assume that there exist t_1 metamorphic testing groups violated in MTG_1 and t_2 metamorphic testing groups violated in MTG_2, if $\frac{t_1}{m} > \frac{t_2}{n}$, we consider MR_1 is more capable than MR_2 in fault detection and MR_1 will be used earlier than MR_2.

As shown in Eq. (1), we then present a score to evaluate the detection capability of MR accordingly. In which, (1) v is the number of violated metamorphic testing groups, that is the number of $g(p^j_{patch}, mtg^l_i) = 0$ in $h(p^j_{patch}, MTG_i) = \wedge^m_{l=1} g(p^j_{patch}, mtg^l_i)$; (2) t is the total number of metamorphic testing groups of MTG_i; (3) s is the evaluation score of each MR in fault detection capability. The larger the score is, the less the number of violated metamorphic testing groups is and the weaker the fault detection capability is. Contrarily, the smaller the score is, the more the number of violated metamorphic testing groups is and the stronger the fault detection capability is. If more than one MRs have identical scores, a tie-breaking strategy [Tu, Xie, Chen et al. (2019)] is used likewise to rank these MRs in their id order. The more sets of $MTGs$ the evaluated patch satisfies, the better the repair quality is.

$$s = 1 - \frac{v}{t} = 1 - \frac{|MTG_i| - \sum^m_{l=1} g(p^j_{patch}, mtg^l_i)}{|MTG_i|} \tag{1}$$

4.2 Execution path coverage driven

Test coverage is a quantitative measurement of the effectiveness of test case in software testing [Rojas, Campos, Vivanti et al. (2015)]. Inspired by this, we present a selection strategy based on path coverage to evaluate the fault detection capability of MR. Because metamorphic relation involves source test cases and follow-up test cases, the coverage of source and follow test cases reflect the capability of MR directly.

As shown in Fig. 1, the set of execution paths of source test suite T_s on patch p^j_{patch} is the source path, denoted as $s - path$, namely $s - path = P_{cov}(p^j_{patch}, T_s)$. The set of execution paths of follow-up test suite T_f on patch p^j_{patch} is the follow-up path, referred to as $f - path$, namely $f - path = P_{cov}(p^j_{patch}, T_f)$. Intuitively speaking, the repetition of $s - path$ and $f - path$ should be as low as possible. That is, the detection capability of MR is strong if source and follow-up test cases cover different execution paths. Fig. 1 presents three scenarios for the measurement of MR detection capability. Fig. 1(a) indicates the most powerful MR that even though the coverages of $s - path$ and $f - path$ are not high respectively, the union of both paths covers all of the execution paths. Fig. 1(b) is a weaker scenario because there exist some intersections between $s - path$ and $f - path$. Fig. 1(c) shows an extremely special scenario. From the perspective of metamorphic relation, Fig. 1(c) and Fig. 1(b) show the identical capability. However, from the perspective of test case, Fig. 1(b) is stronger than Fig. 1(c) since the $s - path$ and $f - path$ in Fig. 1(c) are completely repeated, which seems that the follow-up test cases are redundant and make no contribution to MR detection capability. Therefore, we should take into consideration both intersection and union of $s - path$ and $f - path$ to measure the detection capability of MRs.

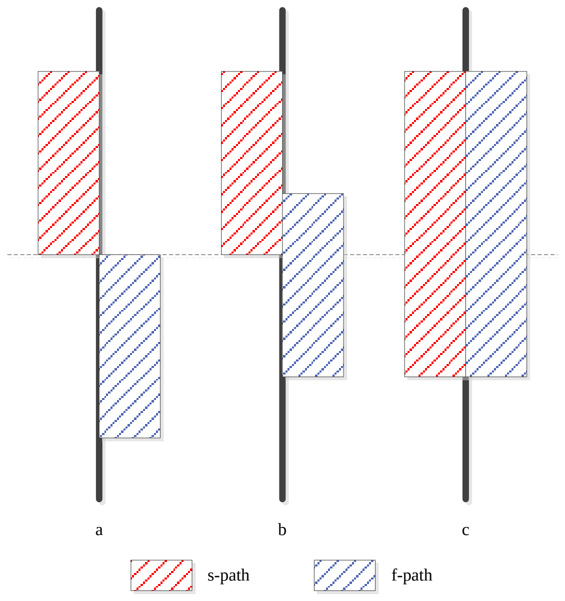

Figure 1: The fault detection capability of metamorphic relation based on execution path coverage

5 Illustration example

5.1 Setup

To evaluate our approach, we use the "mid" program [Wong, Gao, Li et al. (2016)] as the example faulty program which outputs the median among three integers. For example, take 2, 4, 8 as inputs, the output of "mid" program is 4. Take $T_s = \{t_s^1, t_s^2, t_s^3\}$ and $T_f = \{t_f^1, t_f^2, t_f^3\}$ as the source and follow-up test case, o_s and o_f as the source and follow-up outputs. The following three MRs [Jiang, Chen, Kuo et al. (2017b); Wu, Dong, Chen et al. (2017)] are used for "mid" program.

- MR1: Construct T_f from T_s. If $t_f^i = t_s^i + |t_s^i|$, then $o_f = o_s + |o_s|$.
- MR2: Construct T_f by taking the negative of T_s, that is, if $t_f^i = -t_s^i$, then $o_f = -o_s$.
- MR3: Construct T_f by changing the order of any two integers in T_s, then $o_f = o_s$.

As shown in Tab. 1, the faulty statement in program "mid" is s_7 which should be "$m = x$;". $MTG_i (1 \leqslant i \leqslant 3)$ represents the set of metamorphic testing groups for MR_i respectively. $mtg_i^j (1 \leqslant j \leqslant 6)$ is the individual metamorphic testing group in MTG_i which includes source test cases and follow-up test cases generated from the relevant metamorphic relation MR_i. Since we need to compare the repair effectiveness of APR-MT and METARO^3, MTG_i is also used as evaluation test suite to evaluate the repair quality of APR-MT.

Note that, the input MRs for both approaches should contain at least one violated mtg. To construct our comparison, for the approach of APR-MT, we take MR_2 as input to repair "mid" program and produce a program patch, Patch1, when MR_2 is satisfied. For the approach of METARO^3, we take the above three MRs as inputs to generate a program patch, Patch2, when MR_1, MR_2 and MR_3 are satisfied. According to individual evaluation result, satisfy or violate each mtg_i^j, we can evaluate the repair quality of generated patches from two approaches. The more satisfied MRs are, the better the repair quality is.

5.2 Results and analysis

5.2.1 Results

Tab. 1 summarizes the execution information, including coverage of each test case on faulty program and execution results of individual metamorphic testing group on the generated patch. The last two rows of Tab. 1 shows the evaluation results of two patches generated with APR-MT technique and METARO^3 respectively. To evaluate the quality of two patches, we utilize the source test suite and three follow-up test suites as evaluation test suite. "$\checkmark$" implies that the metamorphic testing group is satisfied by the patch, and "$\times$" implies that the metamorphic testing group is violated by the patch. Take the Patch1 row and the 2nd column as example, "$\checkmark$" indicates that Patch1 satisfies mtg_1^1. "$\times$" in cell of the Patch1 row and the 14th column indicates that Patch1 violates mtg_3^1. According to Tab. 1, we have the following observations.

Table 1: Summary of faulty program, input metamorphic relation group and evaluation

Faulty program	MTG_1						MTG_2						MTG_3					
#include ⟨*stdio.h*⟩	mtg_1^1	mtg_1^2	mtg_1^3	mtg_1^4	mtg_1^5	mtg_1^6	mtg_2^1	mtg_2^2	mtg_2^3	mtg_2^4	mtg_2^5	mtg_2^6	mtg_3^1	mtg_3^2	mtg_3^3	mtg_3^4	mtg_3^5	mtg_3^6
void main (int *argc*, char **argv*[]) {	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
int x, y, z, m;	6,6,10	2,4,6	6,4,2	10,10,10	10,6,8	4,2,6	-3,-3,-5	-1,-2,-3	-3,-2,-1	-5,-5,-5	-5,-3,-4	-2,-1,-3	3,5,3	1,3,2	2,3,1	5,5,5	4,3,5	3,1,2
s_1: scanf("%d%d%d",&x,&y,&z);	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●
s_2: $m = z$;	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●
s_3: if ($y < z$) {	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●
s_4: if ($x < y$)	●●	●●			●●	●●	●	●	●		●	●	●	●			●●	●●
s_5: $m = y$;		●●						●	●					●				
s_6: else if ($x < z$)	●●				●●	●●	●				●	●	●				●●	●●
s_7: $m = y$;}	●●					●●	●					●	●				●	●
s_8: else {			●●	●●			●	●	●	●●	●	●	●	●	●●	●●		
s_9: if ($x > y$)			●●	●●			●	●	●	●●	●	●	●	●	●●	●●		
s_{10}: $m = y$;			●●					●	●				●		●			
s_{11}: else if ($x > z$)				●●			●			●●	●	●		●	●	●●		
s_{12}: $m = x$;}							●					●			●			
s_{13}: printf("%d",m);}	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●	●●
Positive/Negative	P	P	P	P	P	P	P	P	P	P	P	N	P	P	P	P	P	N
Generated patches	Satisfaction or violation of three MRs for two generated patches																	
Patch 1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	×	✓	✓	×	×
Patch 2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

(1) Patch1 satisfies MR_1 and MR_2 with the MR_3 is violated, because mtg_3^1, mtg_3^2, mtg_3^5 and mtg_3^6 are violated.

(2) Patch2 satisfies all of the input MRs.

As a result, METARO^3 is more effective than APR-MT because the patch generated from METARO^3 sastifies more metamorphic relations than APR-MT, and it is feasible to improve the repair effectiveness of APR-MT by using several MRs.

5.2.2 Selection strategy

Since METARO^3 takes three MRs as inputs, we should rank them first in descending order according to their fault detection capability. And then examine each MR from top to bottom to find a patch that can satisfy all input MRs.

1) Negative test case driven. Tab. 1 lists the types of metamorphic testing group individually in the 18th row. "P" indicates the positive test case, that is, the mtg_i^j is satisfied by the faulty program. "N" indicates the negative test case, that is, the mtg_i^j is violated by the faulty program. For example, mtg_2^6 is one of the metamorphic testing groups for MR_2 and $mtg_2^6 = (t_{s_2}^6, t_{f_2}^6)$, where $t_{s_2}^6 = (2, 1, 3)$ and $t_{f_2}^6 = (-2, -1, -3)$. By investigating mtg_2^6 on faulty program, the source output $o_{s_2}^6 = 1$ and $o_{f_2}^6 = -2$. MR_2 is violated by faulty program, because $o_{f_2}^6$ should be equivalent to $-o_{s_2}^6$ when $t_{f_2}^6 = -t_{s_2}^6$. Therefore, mtg_2^6 is a negative test case and represented as "N" in Tab. 1.

According to Eq. (3.1) and test case types in Tab. 1, the scores of fault detection capability for three MRs are $Score_1 = 1$, $Score_2 = \frac{5}{6}$ and $Score_3 = \frac{5}{6}$ respectively. MR_1 performs worst among the three MRs, because there is no negative test case in MTG_1. In other words, MR_1 reveals no mistakes in the faulty program and makes no contribution to the faulty detection capability.

The capability of MR_2 and MR_3 is incomparable with their evaluation scores since $Score_2 = Score_3$. Have an insight into the negative test cases mtg_2^6 and mtg_3^6, the buggy program passes both follow-up test cases $t_{f_2}^6$ and $t_{f_3}^6$ while the buggy program fails $t_{s_2}^6$ and $t_{s_3}^6$. Therefore, MR_2 and MR_3 perform equally in this illustration example. We then examine MR_2 first according to the tie-breaking strategy.

2) Execution path coverage driven. As shown in Fig. 2, there exist six execution paths in the faulty "mid" program. The nodes in Fig. 2 indicate the statements in program. $P_i(1 \leqslant i \leqslant 6)$ represents the i^{th} execution path across the relevant nodes. The coverage information of each statement of corresponding test case examined on faulty program is reported in Tab. 1. "•" indicates the statement is executed by the test case; "• •" represents that the statement is executed by the source and follow-up test case; "• " implies that the statement is only executed by source test case, while " •" suggests that the statement is only executed by follow-up test case. Take the s_4 row and the 8th column as example, "• " indicates that statement s_4 is executed by the source test case $t_{s_2}^1$ but not executed by the follow-up test case $t_{f_2}^1$.

Following the definition of $s-path$ and $f-path$ in Section 4.2 and coverage information in Tab. 1, the $s-path$ and $f-path$ for three MRs are summarized as follows.

$s-path$=$\{P_2, P_1, P_4, P_5, P_3, P_2\}$=$\{P_2, P_1, P_4, P_5, P_3\}$;

$f-path_1$=$\{P_2, P_1, P_4, P_5, P_3, P_2\}$=$\{P_2, P_1, P_4, P_5, P_3\}$;

$f-path_2$=$\{P_5, P_4, P_1, P_6, P_6, P_5\}$=$\{P_5, P_4, P_1, P_6\}$;

$f-path_3$=$\{P_4, P_6, P_5, P_6, P_2, P_3\}$=$\{P_4, P_6, P_5, P_2, P_3\}$;

For MR_1, $s-path \cup f-path_1$ covers all of the six execution traces, and $s-path$ is absolutely identical with $f-path_1$ which follows the case of Fig. 1(c), therefore MR_1 is the worst metamorphic relation in fault detection capability. For MR_2, $s-path \cup f-path_2 = \{P_1, P_2, P_3, P_4, P_5, P_6\}$ and $s-path \cap f-path_2 = \{P_1, P_4, P_5\}$, which is the case of Fig. (1).b. Hence, MR_2 performs better than MR_1. For MR_3, $s-path \cup f-path_3 = \{P_1, P_2, P_3, P_4, P_5, P_6\}$ and $s-path \cap f-path_3 = \{P_2, P_3, P_4, P_5\}$. In accordance with the argument of execution path coverage driven strategy, the $s-path \cap f-path$ should be as small as possible and the relevant MR performs well. Since $|s-path \cup f-path_2| = |s-path \cup f-path_3|$ and $|s-path \cap f-path_2| < |s-path \cap f-path_3|$, MR_2 outperforms MR_3. Thus, MR_2 performs best and MR_1 performs worst in fault detection capability. This helps debuggers to choose MR_2 to inspect first.

5.2.3 Analysis

The two patches generated from one MR and three MRs show distinct evaluation results. Patch2 satisfies all the three MRs, while Patch1 satisfies MR_1 and MR_2 and violates MR_3, which demonstrates our proposition that the patch generated from one MR is plausible. The repair effectiveness, especially repair quality, is improved significantly by applying more

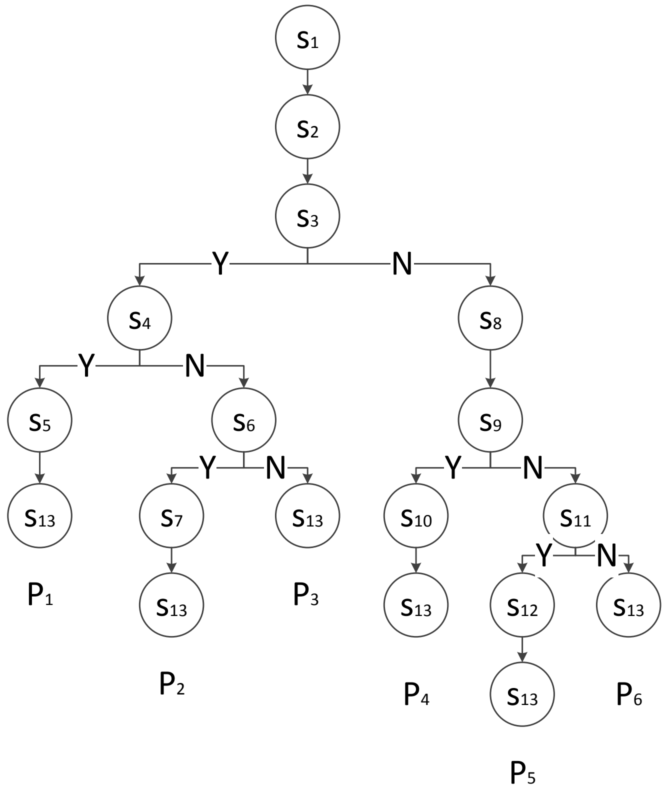

Figure 2: The execution paths of "mid" program

than one MRs to guide the repair and evaluation process. Therefore, METARO^3 is superior to APR-MT.

According to the ranking results of three MRs with two selection strategies, both strategies can identify the same ranking lists with MR_2 ranks in the first place and MR_1 ranks last. That is, MR_2 performs best while MR_1 performs worst in fault detection capability. However, negative test case driven strategy seems inferior to execution path coverage driven strategy since MR_2 has the same performance to MR_3 with the former one, while MR_2 is more powerful than MR_3 with the later one. This is reasonable that both source and follow-up test cases determines execution results, which affects the effectiveness of selection strategies. The impact of test cases will be discussed in depth in Section 6.

6 Threats to validity

6.1 Metamorphic relations

The key insight of our approach is the metamorphic relation group which contains more than one MRs. We consider MR_3 is not implementable to expose errors in the faulty program even though both mtg_3^1, mtg_3^2, mtg_3^5 and mtg_3^6 cover the faulty statement s_7.

Additionally, as a strategy of test case generation [Chen, Cheung and Yiu (1998)], MR is also used to generate follow-up test cases along with source test cases, which affects the execution path coverage of $f - path$. Hence, it is worthwhile to identify efficient, diverse and nonredundant MRs for METARO^3 in the future work.

6.2 Source test cases

Since follow-up test cases are generated from source test cases and relevant MR, source test cases affect not only source execution path $s - path$ but also follow-up execution path $f - path$ and affect the execution result of mtg, satisfy or violate consequently. For example, the source test case (5, 5, 5) in mtg_1^4, mtg_2^4 and mtg_3^4 is considered as a ineffective source test case, because all of the follow-up paths are the same as source paths and three MRs are satisfied by faulty program when executing the three mtg_s. Therefore, (5,5,5) cannot reveal any mistakes in faulty program and benefit the detection effectiveness of MRs. Selection of good source test cases with impractical test cases wiped off will undoubtedly increase the effectiveness of METARO^3.

7 Related work

Over the past decades, various techniques have been proposed to advance the effectiveness of program repair in terms of repair quality and repair cost (e.g., patch overfitting, test case reduction, search space constrain).

Oliveira et al. [Oliveira, Souza, Le Goues et al. (2016)] propose three crossover operators to explore new repairs for buggy program under repair. OP1Space generates two offspring by swapping the tail of one subspace chosen randomly. Unif1Space generates a binary mask for the chosen subspace, which actually is the index to be used in two parent representation spaces. OPAllS swaps the larger part of parent representation instead of swapping the tail of single space of OP1Space. In their recent work [Oliveira, de Souza, Le Goues et al. (2018)], the mutation operator, Subspace Mutation, is used to change one point of the random chosen space and produce a new edit for buggy program.

Hua et al. [Hua, Zhang, Wang et al. (2018)] present a novel technique, SketchFix, which transforms the faulty program to a sketch based on AST node-level schemas that represents all candidate patches of same schemas. Debuggers then compile and execute only one sketch rather than hundreds of candidate patches on tests. Thus, SketchFix can reduce the repair space and the amount of compile and execution times considerably.

Xiong et al. [Xiong, Liu, Zeng et al. (2018)] check the patch correctness by investigating the similarity of test case executions. The failing test cases are likely to behave differently on buggy and fixed programs, which help to filter out some incorrect patches.

Yi et al. [Yi, Ahmed, Karkare et al. (2017)] integrate program repair with intelligent tutoring system for programming to increase the repair rate of programs submitted by introductory programming student and professional developer.

Yu et al. [Yu, Martinez, Danglot et al. (2017)] present two test case generation approaches with purpose of alleviating overfitting problem for search-based and semantics-based program repair, namely MinImpact and UnsatGuided respectively.

FootPatch [van Tonder and Le Goues (2018)], a static program repair technique, is implemented by utilizing Separation Logic to repair programs of pointer safety properties with the pre-specified semantic effects instead of test case execution.

8 Conclusion and future work

Program repair is a labor-intensive activity in software testing. Automatic program repair is proposed for the automation of fault localization, bug fix and patch evaluation procedures. Test suite based APR techniques produce patches with the existence of test oracles. Jiang et al. [Jiang, Chen, Kuo et al. (2017b)] integrate metamorphic testing with test suite based APR to deliver a patch without the need of test oracles. One of the challenges in APR-MT is the repair quality of generated patches, because APR-MT takes only one MR to repair faulty program and evaluate generated patches.

To address this challenge, we propose to use multiple MRs which should be hard satisfied by generated patches, and these MRs are ordered with their fault detection capability to speed up the repair process. According to the illustration example, our approach is demonstrated to produce program patch of higher quality than that of APR-MT. Moreover, we evaluate the generated patch with not only the input MTGs in MR level but also the source and follow-up test cases in test case level, which provides a more precise assessment for the effectiveness of METARO^3 and the quality of generated patches.

Acknowledgement: The work was supported by a grant from National Natural Science Foundation of China (No. 61772423).

References

Barr, E. T.; Harman, M.; McMinn, P.; Shahbaz, M.; Yoo, S. (2015): The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507-525.

Chen, T. Y.; Cheung, S. C.; Yiu, S. M. (1998): Metamorphic testing: a new approach for generating next test cases. *Technical report, Technical Report HKUST-CS98-01*. Department of Computer Science, Hong Kong University.

Chen, T. Y.; Kuo, F. C.; Liu, H.; Poon, P. L.; Towey, D. et al. (2018): Metamorphic testing: a review of challenges and opportunities. *ACM Computing Surveys*, vol. 51, no. 1, pp. 1-27.

Gazzola, L.; Micucci, D.; Mariani, L. (2017): Automatic software repair: a survey. *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 43-67.

Gupta, R.; Pal, S.; Kanade, A.; Shevade, S. (2017): Deepfix: fixing common c language errors by deep learning. *Thirty-First AAAI Conference on Artificial Intelligence*, pp. 1345-1351.

Hua, J.; Zhang, M.; Wang, K.; Khurshid, S. (2018): Sketchfix: a t ool f or automated program repair approach using lazy candidate generation. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 888-891.

Jiang, M.; Chen, T. Y.; Kuo, F. C.; Ding, Z.; Choi, E. H. et al. (2017): A revisit of the integration of metamorphic testing and test suite based automated program repair. *IEEE/ACM 2nd International Workshop on Metamorphic Testing*, pp. 14-20.

Jiang, M.; Chen, T. Y.; Kuo, F. C.; Towey, D.; Ding, Z. (2017): A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software*, vol. 126, pp. 127-140.

Kaleeswaran, S.; Tulsian, V.; Kanade, A.; Orso, A. (2014): Minthint: automated synthesis of repair hints. *Proceedings of the 36th International Conference on Software Engineering*, pp. 266-276.

Kong, X.; Zhang, L.; Wong, W. E.; Li, B. (2018): The impacts of techniques, programs and tests on automated program repair: an empirical study. *Journal of Systems and Software*, vol. 137, pp. 480-496.

Le Goues, C.; Dewey Vogt, M.; Forrest, S.; Weimer, W. (2012): A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. *34th International Conference on Software Engineering*, pp. 3-13.

Le Goues, C.; Holtschulte, N.; Smith, E. K.; Brun, Y.; Devanbu, P. et al. (2015): The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236-1256.

Le Goues, C.; Nguyen, T.; Forrest, S.; Weimer, W. (2012): Genprog: a generic method for automatic software repair. *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54-72.

Liu, K.; Koyuncu, A.; Kim, K.; Kim, D.; Bissyande, T. F. D. A. (2018): Lsrepair: live search of fix ingredients for automated program repair. *25th Asia-Pacific Software Engineering Conference*.

Liu, X.; Zhong, H. (2018): Mining stackoverflow for program repair. *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 118-129.

Liu, Y.; Zhang, L.; Zhang, Z. (2018): A survey of test based automatic program repair. *Journal of Software*, vol. 13, no. 8, pp. 437-453.

Long, F.; Rinard, M. (2015): Staged program repair with condition synthesis. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 166-178.

Long, F.; Rinard, M. (2016): An analysis of the search spaces for generate and validate patch generation systems. *IEEE/ACM 38th International Conference on Software Engineering*, pp. 702-713.

Long, F.; Rinard, M. (2016): Automatic patch generation by learning correct code. *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 298-312.

Martinez, M.; Monperrus, M. (2016): Astor: a program repair library for java (demo). *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 441-444.

Martinez, M.; Monperrus, M. (2019): Astor: exploring the design space of generate-and-validate program repair beyond genprog. *Journal of Systems and Software*, vol. 151, pp. 65-80.

Mechtaev, S.; Nguyen, M. D.; Noller, Y.; Grunske, L.; Roychoudhury, A. (2018): Semantic program repair using a reference implementation. *Proceedings of the 40th International Conference on Software Engineering*, pp. 129-139.

Mechtaev, S.; Yi, J.; Roychoudhury, A. (2016): Angelix: scalable multiline program patch synthesis via symbolic analysis. *Proceedings of the 38th International Conference on Software Engineering*, pp. 691-701.

Nguyen, H. D. T.; Qi, D.; Roychoudhury, A.; Chandra, S. (2013): Semfix: program repair via semantic analysis. *35th International Conference on Software Engineering*, pp. 772-781.

Nguyen, T. H. (2014): *Automating Program Verification and Repair Using Invariant Analysis and Test Input Generation*. PhD thesis, The University of New Mexico.

Oliveira, V. P. L.; de Souza, E. F.; Le Goues, C.; Camilo-Junior, C. G. (2018): Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering*, pp. 1-27.

Oliveira, V. P. L.; Souza, E. F.; Le Goues, C.; Camilo-Junior, C. G. (2016): Improved crossover operators for genetic programming for program repair. *International Symposium on Search Based Software Engineering*, pp. 112-127.

Qi, Y.; Mao, X.; Lei, Y. (2013): Efficient automated program repair through fault-recorded testing prioritization. *IEEE International Conference on Software Maintenance*, pp. 180-189.

Rojas, J. M.; Campos, J.; Vivanti, M.; Fraser, G.; Arcuri, A. (2015): Combining multiple coverage criteria in search-based unit test generation. *International Symposium on Search Based Software Engineering*, pp. 93-108.

Roychoudhury, A. (2016): Semfix and beyond: semantic techniques for program repair. *Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems*, pp. 2-2.

Saha, R. K.; Lyu, Y.; Yoshida, H.; Prasad, M. R. (2017): Elixir: effective object-oriented program repair. *32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 648-659.

Tu, J.; Xie, X.; Chen, T. Y.; Xu, B. (2019): On the analysis of spectrum based fault localization using hitting sets. *Journal of Systems and Software*, vol. 147, pp. 106-123.

van Tonder, R.; Le Goues, C. (2018): Static automated program repair for heap properties. *IEEE/ACM 40th International Conference on Software Engineering*, pp. 151-162.

Weimer, W.; Fry, Z. P.; Forrest, S. (2013): Leveraging program equivalence for adaptive program repair: models and first results. *2 8th I EEE/ACM I nternational Conference on Automated Software Engineering*, pp. 356-366.

Weiss, A.; Guha, A.; Brun, Y. (2017): Tortoise: interactive system configuration repair. *32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 625-636.

Wong, W. E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F. (2016): A survey on software fault localization. *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707-740.

Wu, T.; Dong, Y.; Chen, T. Y.; Jiang, M.; Lau, M. et al. (2017): Integration of metamorphic testing with program repair methods based on adaptive search strategies and program equivalence. *International Conference on Formal Engineering Methods*, pp. 413-429.

Xiong, Y.; Liu, X.; Zeng, M.; Zhang, L.; Huang, G. (2018): Identifying patch correctness in test-based program repair. *Proceedings of the 40th International Conference on Software Engineering*, pp. 789-799.

Xiong, Y.; Wang, J.; Yan, R.; Zhang, J.; Han, S. et al. (2017): Precise condition synthesis for program repair. *IEEE/ACM 39th International Conference on Software Engineering*, pp. 416-426.

Xuan, J.; Martinez, M.; Demarco, F.; Clement, M.; Marcote, S. L. et al. (2017): Nopol: automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34-55.

Yang, J.; Zhikhartsev, A.; Liu, Y.; Tan, L. (2017): Better test cases for better automated program repair. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 831-841.

Yi, J.; Ahmed, U. Z.; Karkare, A.; Tan, S. H.; Roychoudhury, A. (2017): A feasibility study of using automated program repair for introductory programming assignments. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 740-751.

Yu, Z.; Martinez, M.; Danglot, B.; Durieux, T.; Monperrus, M. (2017): Test case generation for program repair: a study of feasibility and effectiveness. *arXiv:1703.00198*.