Article

Modification Point Aware Test Prioritization and Sampling to Improve Patch Validation in Automatic Program Repair

Yazhini Venugopal ¹, Phung Quang-Ngoc ¹ and Lee Eunseok ^{2,*}

¹ Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, Korea; yazhiniv@skku.edu (Y.V.); ngocpq@skku.edu (P.Q.-N.)

² College of Software, Sungkyunkwan University, Suwon 16419, Korea

* Correspondence: leees@skku.edu

Received: 30 December 2019; Accepted: 22 February 2020; Published: 27 February 2020

Abstract: Recently, Automatic Program Repair (APR) has shown a high capability of repairing software bugs automatically. In general, most of the APR techniques require test suites to validate automatically generated patches. However, the test suites used for patch validation might contain thousands of test cases. Running these whole test suites to validate every program variant makes the validation process not only time-consuming but also expensive. To mitigate this issue and to enhance the patch validation in APR, we introduce (1) MPTPS (Modification Point-aware Test Prioritization and Sampling), which iteratively records test execution. Based on the failed test information, it performs test prioritization, then sampling to reduce the test execution time by moving forward the test cases that are most likely to fail in the test suite; and (2) a new fitness function that refines the existing one to improve repair efficiency. We implemented our MPPEngine approach in the Astor workspace by extending jGenProg. And the experiments on the Defects4j benchmark against jGenProg show that, on average, jGenProg consumes 79.27 s to validate one program variant, where MPPEngine takes only 33.70 s for results in 57.50% of validation time reduction. Also, MPPEngine outperforms jGenProg by finding patches for six more bugs than jGenProg.

Keywords: Automatic Program Repair (APR); test prioritization; test sampling; fitness function; patch validation; patch assessment

1. Introduction

The increase in software complexity often results in high debugging and maintenance costs. A recent study on software depicted that the debugging process of the complex software is time-consuming and tedious. In addition, it takes up to 25–50% of overall project expenses [1–3]. A study by Capgemini [4] indicates that testing is not as efficient as it should be, and it is an important objective of any quality program. They also stated that QA and testing efforts have reduced by about 26% this year (2019), but in the next two years, it will again go up to 30%. Bug fixing is also one of the factors that cause this high expense [5]. Usually, the developer or tester finds the fault(s) in the program and fixes it manually. Therefore, it requires time, effort, and manpower to find and fix issues in the software. To reduce manual efforts in testing to overcome this problem, Automatic Program Repair (APR) has been introduced. APR provides a solution to the buggy program by debugging the faults automatically without any human intervention. So, APR plays a vital role in automated debugging [6]. There are two main approaches to APR. (1) Generate and Validate approach: it modifies the original buggy program by applying a set of change operators. As the name suggests, it generates the patch and validates it against the test suite to check patch correctness; and (2) Semantic-driven

approach: it formally or explicitly encodes the buggy program, converts into a formula that produces solutions that are expected to fix the fault [7]. These APR tools come with an inbuilt fault localization tool. Initially, the fault localizer will be fed with a buggy program and a test suite related to that program as an input. It identifies suspicious statements in the program and each suspicious statement will be given a suspicious score. Then the patch is generated using the Generate and Validate approach or Semantic-driven approach. Once a patch has been generated, the validation process takes place. In the validation process, the correctness of the patch is evaluated by using test cases to ensure that the patch fixes the fault and is not introducing any new or additional issue(s). If the patch passes all the test cases in the test suite, then it is called a valid patch or test suite adequate patch. Otherwise, it is considered as an incorrect patch.

Unfortunately, chances are that the validation test suite might include a large number of test cases. Most of the test cases consume a lot of time for test execution, which requires time and cost. As Rothermel et al. [8] reported in their paper, one of their products took seven weeks to execute the entire test suite against a software bug, which is a good example of a long running test suite problem. Jurgens et al. [9] discussed the long running and large test suites impact on testing in their paper and provided a solution by introducing test prioritization based on code changes. On the other hand, APR produces a lot of invalid patches that should be identified and filtered out using test suites in patch validation at the early stage itself so that APR could generate more program variants in the given time. In case the patch encounters a large test suite, it has to spend a significant amount of time to validate the program variants until it produces a test suite adequate patch. Sometimes, without executing the entire test suite, the patch validation process ends due to time out where we cannot trace the failing test cases even if there are some. To alleviate this problem, we plan to improve the patch validation process in the APR by identifying invalid patches in the early stage of patch validation through fault-based test case prioritization. Thus, it speeds up the whole validation process in an effective way. Previous techniques run all the test cases in the test suite even if it fails a few test cases in the suite. And, the number of failing test cases count is used to calculate the fitness score. However, in our approach, we decided to implement test case sampling and run test cases in small subsets with an equal number of test cases. Test case sets were executed one by one until it encounters a failure with the set. Based on the test case failing count and executed test case information, we present a new fitness function. This will help in improving the repair efficiency of our approach.

Enhancing the patch validation process in APR is a key step for producing a more reliable patch in less time. This paper addresses this goal with the following contributions,

- We implement MPTPS (Modification Point-aware Test Prioritization and Sampling) to reduce test execution time.
- A new enhanced fitness formula to regulate the already available fitness function in Astor.
- Finally, experiments of the proposed method conducted in the Defects4j benchmark, results compared against jGenProg, and answers to the research questions.

The rest of this paper is arranged as follows. Section 2 provides information about the related works to this research. In Section 3, we explain background concepts that need to be understood before the proposed method. Section 4 introduces our proposed approach, MPPEngine, including MPTPS, and the new fitness function in detail. Section 5 presents our experimental evaluation with research questions and the relevant materials have been provided in Supplementary Materials. Results and discussions have been elaborated in Section 6. At last, in Section 7, we conclude this paper by summarizing our proposed approach.

2. Related Works

In recent years, there have been several techniques and tools proposed for Automatic Program Repair (APR). GenProg is one of the first APR tools which uses genetic programming. It is a state of

the art tool developed for C program automated repair. GenProg generates patches using the genetic algorithm through the iterative process [10].

First, fault localization takes place to locate faults. After that, GenProg produces candidate solutions using the mutation process (atomic changes), and then the test suite is used to validate them. These two processes run iteratively until it finds a valid patch. With respect to this, many kinds of research took place by extending GenProg [10]. Martinez et al. [11–13] proposed jGenProg as part of Astor, which is a java version of GenProg. Java programs can be repaired automatically with the help of jGenProg. Consequently, it is validated on the buggy programs from the Defects4j dataset [14]. GenProg and jGenProg take a lot of time for validating a patch due to long-running test cases. Qi et al. [15] presented a solution to this problem by introducing Function-based Part Execution where they validate only part of the program instead of validating the whole program. Yang et al. [16] introduced a framework to detect overfit or incorrect patches by generating test cases using fuzz testing. This framework is called OPAD, and it effectively filtered incorrect patches.

Test case prioritization and sampling is a widely researched area in regression testing [17]. However, in the case of APR, there are not many pieces of research conducted in prioritization and sampling. Qi et al. [18] proposed a tool named TrpAutoRepair that includes Fault Recorded Testing Prioritization (FRTTP) to reduce the cost of testing. TrpAutoRepair automatically generates patches for C Program, and it improves efficiency by reducing test case execution. It was the first time introducing insights of prioritization in automated program repair. Their proposed method ranks the test cases based on the test case failing count. Every time, the same test suite is updated, prioritized, and validated against all the program variants generated by GenProg. In other words, regardless of the modification point chosen, it follows the prioritized order of previous execution. Therefore, to overcome this problem, we have proposed modification point aware prioritization where a different prioritized test suite is maintained for every program variant based on modification point information. Again, Qi et al. [19] used classic prioritization techniques [18,20] in RSRepair to identify the invalid patch early in the validation process. In both approaches, the same prioritized test suite was used for all the program variants.

Jang et al. [21] introduced a tool AdqFix based on variants by extending the Astor environment. They proposed a new fitness function to enhance the repair efficiency of the existing fitness function. Fast et al. [22] proposed enhancements for fitness functions in terms of efficiency and precision. They implemented test case sampling, but it does not ensure the functionality as the dynamic predicates used to collect information required for the fitness function. De Souza et al. [23] introduces a fitness function with the help of the program/source code check-points to differentiate the individuals with the same fitness score.

3. Background

3.1. Automatic Program Repair

Automatic Program Repair (APR) is a technique that automatically generates patches to fix the buggy program. Figure 1 shows an overview of APR in detail. Generally, APR tools have three main phases: Fault Localization, Program Modification, and Program Validation. In Astor, the GZoltar [24] fault localization tool with the Ochiai [25] algorithm is used. Initially, the Fault localization tool takes Faulty Program and Repair Tests as inputs and automatically finds suspicious statements in the program. Each suspicious statement is given a suspicious score and selected randomly to undergo modification in the Program Modification phase. It uses a genetic algorithm to generate candidate solutions, subsequently, by applying atomic operators. Candidate solutions are generated based on the given number of generations or time limit. These generated candidate solutions are then applied to the buggy program to produce patches. These candidate solutions are also known as program variants.

Figure 1. Automatic Program Repair (APR): Generate and Validate Approach.

Soon after the generation, the program variants will be validated in the Program Validation phase against the Repair Tests (patch validation test suite) used during Fault Localization. If all the test cases get passed, it outputs a test suite adequate patch(es), also known as Repaired Programs. Otherwise, it is considered an invalid patch, and it is dropped or sent for modification once again based on its fitness score.

Generate and Validate Approach

The Generate and Validate approach is an important technique in automatic program repair. This approach executes an iterative process that contains two main components, generate and validate. GenProg is one of the first APR tools introduced using the Generate and Validate approach that is based on genetic programming [10]. Based on the higher suspicious score, the generate component selects the locations where the modification should take place, and it uses change operators to modify the buggy program and produces candidate solutions (program variants). In some previous techniques, change operators not only applied to the buggy program but also to the candidate solutions [7,10]. Searching of a candidate solution ends when all the possible elements in the search space have been considered or when the time allocated to the repairing process is expired. The validation component checks the correctness of the program variants (candidate solutions) that have been generated. Currently, most of the available generate and validate techniques evaluate the candidate solution correctness by running the available test suite. During validation, some or all candidate solutions that fail many test cases are usually discarded. However, if the candidate solution passes against all the available test cases, it is considered a test suite adequate patch and is delivered as a possible fix to the developer.

3.2. Test Case Prioritization and Sampling

Rothermel et al. [8] defined the Test Case Prioritization Problem in the following way:

Given: T , a test suite; PT , the set of permutations of T ; f , a function from PT to the real numbers.
 Problem: Find $T' \in PT$ such that $(\forall T'') (T'' \in PT) (T'' \neq T') [f(T') \geq f(T'')]$.

Here, PT represents the set of all possible permutations (in terms of orderings) of T , a specified test suite; f is a transition function that can evaluate an awarded value for any ordering of PT . There are many techniques available for test prioritization, such as search-based, coverage-based, risk-based, fault-based, and so on [17]. Here in this paper, we use fault-based prioritization. Fast et al. [22]

implemented random sampling that chooses test cases randomly in a uniform way and time-aware test-suite reduction (introduced by Walcott et al.) that uses a genetic algorithm to reorder the test suite based on testing time constraints. Both have a high chance of including test cases that are not related to the modification point (program variant) or the test cases that are least likely to fail. The base work to our modification point-aware prioritization is that of Qi et al. [18,19] who proposed the FRTP (Fault-Recorded Testing Prioritization) technique in the GenProg environment and RSRepair. These are the only available papers on prioritization in APR (which we have searched using keywords and confirmed). The reason we chose this technique as the base work is that the Fault-Recorded Prioritization technique does not require any previous execution of test cases. Rather, it iteratively extracts test case execution information during the repair process. Then, their proposed method ranks the test cases based on the failing test cases, which helps in early incorrect patch detection. Every time the same test suite is updated, prioritized, and validated against all the program variants generated by GenProg [10], RSRepair [19]. In other words, regardless of the modification point chosen, it follows the prioritized order of previous execution. Therefore, to overcome this problem, we have proposed a modification point-aware prioritization where a different prioritized test suite is maintained for every program variant based on modification point information.

4. Proposed Method

Patch validation in APR is often time-consuming due to large test suites and (or) long-running test cases. Running such test suites are considered expensive and time-consuming. It becomes more serious when the software program is large and complex. Also, the fitness function is calculated by adding up the number of failed test cases count in Astor. The objective of our approach is to reduce the time and cost of patch validation in APR by prioritizing and sampling the validation test suite for every program variant with the help of modification point information. Also, to present a new fitness function to improve repair efficiency. Although many kinds of research have been going on for prioritization in regression testing, it is not common in APR patch validation. Our approach will help to identify invalid patches sooner that let the APR tool generate more solutions in the given time. Therefore, our contributions in this paper help in reducing the test execution time and the cost of validation while improving the repair effectiveness and efficiency.

4.1. Modification Point-Aware Test Prioritization and Sampling (MPTPS)

Figure 2 clearly explains the overview of our proposed method MPTPS (Modification Point-aware Test Prioritization and Sampling). Initially, *Test Runner* takes *Program Variant* and *Ordered Test Cases* from *Test Suite* as inputs for validation and produces a result. Ordered Test Cases are nothing but prioritized and then sampled test cases (in the same prioritized order). From the *Test Results*, we extract the failing test cases information and *Update MPPTable* by adding (increasing) value '1' to every failing test case in the patch killing count column of the table. After *MPPTable* is updated with the test execution results, *Test Case Prioritization* takes place by prioritizing the patch killing count in descending order from the table. Prioritized test cases then undergo the *Test Cases Sampling* phase to split the test suite into multiple subsets. These *Ordered Test Cases* subsets are executed one by one until the last subset, if there are no failing tests during execution. If all the subsets (test cases) pass, it is delivered as a *Test Suite Adequate Patch* to the developer. Otherwise, the *Test Runner* will be stopped after completely executing the subset with a failing test case(s). In this case, the outcome is considered a *Failed Patch*. Subsequently, the fitness score is calculated for the patch and based on the score, and the *Failed Patch* might undergo modification once again.

Figure 2. Modification Point-aware Test Prioritization (MPTPS) overview.

4.1.1. Modification Point-Aware Test Prioritization—MPPTable

We implemented the modification point aware prioritization by extending the FRTP approach [18]. During validation, for every program variant, modification point information, related test cases, and its failing count (known as patch killing count) are mapped together in a table format. Patch killing count is nothing but a number of times a test case fails a program variant. By monitoring the test cases that make a program variant fail, we record that information to calculate the patch killing count of a test case. These details will be updated and stored in the table (MPPTable) for each program variant. The steps are as follows,

- MPPTable is updated (based on the test results) every time a new program variant is generated for the modification point(s).
- Test cases in the table are prioritized based on the patch killing count of the test case in a descending order, i.e., placing the test cases with largest to smallest count.
- If the test cases have a similar patch killing count, it will be ordered randomly among themselves.

Therefore, for every program variant, a separate prioritized test suite is maintained.

4.1.2. Test Case Sampling

In order to reduce the cost of test case execution, test cases that are prioritized (ordered) will be executed selectively. To do this, the test cases in the prioritized test suite will be split into subsets.

Each subset in the test suite will have an equal number of test cases that will be set before the execution. If all the test cases in the first subset passed against the program variant, the next set will be executed. And the process continues until the last subset, in case there are no failed test cases found during execution. If the number of failed test cases is zero, the patch is chosen as a test-suite adequate patch. Otherwise, it is considered an incorrect patch. In the case of an incorrect patch, if a subset has any failed test case(s), it will not stop the testing process right away, but the remaining test cases in the chosen subset will be executed completely. As for the valid patch, we run all the test cases in the test suite, and only for the incorrect patch, we stop the execution during execution. The failed test cases count and executed test case count are used in the new fitness function calculation.

4.2. New Fitness Function

The fitness function is used to evaluate the patch using the output from the validation process. This helps to identify whether the patch is a solution or not. Also, the incorrect patch with the best fitness value (low value) will undergo the repairing process once again. So, there is a chance of producing a test adequate patch during the re-repairing process.

In Astor [11–13], whenever a patch encounters a failed test case, it sums up the failed test case count and sets it as a fitness value. If there are no failed test cases, the patch is considered to be a test adequate patch. Following the same fitness function in our approach will not be accurate and efficient as we run test cases in subsets from the prioritized test suite. For example, in our approach, one failed test case out of 50 test cases is different from one fail test case out of 100 test cases. So, we are presenting a new concrete fitness function to alleviate the fitness function efficiency. The fitness function will be calculated based on the following formula,

$$F_s = \frac{T_E}{T_T} - \frac{T_F}{T_E} \quad (1)$$

where,

T_T —Total Test Cases Count

T_E —Executed Test Cases Count

T_F —Failed Test Cases Count

Condition. Let there exist two patches: p_1 , p_2 , which have the same number of test cases. Patch ‘ p_1 ’ provides the best fitness, if $p_1(T_E) > p_2(T_E)$ and $p_1(\frac{T_F}{T_E}) < p_2(\frac{T_F}{T_E})$.

By calculating the fitness score using the above formula, the efficiency of fitness scores can be maintained. As the most likely to fail test cases are expected to be present in the initial subsets of the prioritized test suite, calculating fitness based on this formula would help in maintaining and improving repair efficiency.

5. Experimental Evaluation

In this section, we describe our experimental setup, results, and address research questions for the proposed approach MPPEngine.

5.1. Experimental Setup

Our proposed approach MPPEngine is implemented by extending jGenProg in the Astor [11–13] workspace. As we evaluated our approach MPPEngine against jGenProg, we used the same Defects4j [14] benchmark that was used to evaluate jGenProg in the Astor [11–13] workspace. The Defects4j [14] benchmark contains 6 subjects in total, but 2 of the subjects do not come with JUnit test cases. Therefore, we conducted our experiments on 4 subject programs (Chart, Lang, Math, and Time) with 224 bugs from the Defects4j benchmark, as mentioned in Table 1.

Table 1. Defects4j benchmark for experiments [14].

Subjects	No. of Buggy Programs	KLoc *	Test KLoc *	Tests
JFreeChart	26	96	50	2205
Commons Lang	65	22	6	2245
Commons Math	106	85	19	3602
Joda Time	27	28	53	4130

* KLoc for the most recent version, as reported by SLOC Count.

In Astor, the fault localization tool used is GZoltar [24], and it comes with the Ochiai algorithm [25]. As we extended jGenProg in Astor to implement MPPEngine, we used the same in our approach as well. We ran experiments on a virtual machine with a Ubuntu OS, Intel Core-i5 processor, CPU with @3.50 GHz, and RAM of 4GB. For all the experiments, the time limit was set to 3 h (i.e., 180 min), the total generation set to 10,000 generations in maximum, and sampling size to 20 (each sampled subset will have 20 test cases). In other words, MPPEngine tries to generate a valid patch within 3 h of the time limit or within 10,000 generations running the test suite in subsets each with 20 test cases. Also to reduce randomness, we ran all the experiments 3 times and took the average value for the results.

5.2. Research Questions

The evaluation of our approach is accompanied by a set of research questions that mainly focus on the improvement achieved by our proposed method against jGenProg. In this section, we are going to explain the importance of the research questions that we have included and their role in the experiment findings.

- RQ1 [Validation Time]: To what extent is our proposed approach effective in reducing test execution time (cost) compared with jGenProg?

In particular, we discuss the performance difference between jGenProg and MPPEngine in terms of validation time reduction. Here, we measure the average time taken by each technique to validate one patch. We tabulate Min, Median, Max, SD, and Average time taken by the approaches. Finally, we compare the results and prove our approach is more effective than the existing approach.

- RQ2 [Repair Efficiency]: Does MPPEngine produce more patches for bugs compared to jGenProg?

With the implementation of MPTPS, our approach should validate more program variants in the given time and produce more plausible patches compared to jGenProg because MPTPS helps in early detection of incorrect patches and reduces validation time and validates more patches in the remaining time. In RQ2, we investigate the repair efficiency of each technique and prove our MPPEngine approach is equally or more efficient than jGenProg.

- RQ3 [Repair Effectiveness]: Is there any new patch generated for bugs by MPPEngine for which jGenProg does not produce any solution?

We have implemented a technique that includes fault-recorded test prioritization and sampling to identify failing test cases in the early stage of validation. And based on the results, we calculate a new fitness function. Fitness function is an important factor for selecting the best candidates among incorrect patches for re-repairing. There is a chance that APR tools produce test suite adequate patches after re-modification. So, under this research question, we will be experimenting with the repair effectiveness of our approach by monitoring patch generation for each bug.

6. Results and Discussion

6.1. RQ1: To What Extent Is Our Approach Effective in Reducing Test Execution Time (cost) Compared to jGenProg?

We investigated the effectiveness of MPPEngine against jGenProg in terms of reducing test execution time and cost. We ran experiments on the Defects4j [14] dataset used in Astor [11–13] to compare the results more effectively. So, we filtered out 48 bugs from 4 Defects4j (Chart, Math, Lang, and Time) subjects for which jGenProg said to have generated at least one test suite adequate patch. As per the experiments conducted among other tools in Astor [11–13], jGenProg generated patches faster using test suites. There were 19 patches out of 48 patches generated within 3 h. So, we decided to check that condition with our approach. Randomness in the results was reduced by running all the experiments 3 times.

In terms of test case reduction with respect to time, MPPEngine is better for 42 bugs (42/48) and jGenProg is better for 6 bugs (6/48). Table 2 depicts that Minimum time to validate a program variant in MPPEngine is 0.86 s where jGenProg is 4.80 s. Meanwhile, Median and Maximum time consumed to validate a patch in MPPEngine is lesser (19.53 s, 119.75 s) than jGenProg (30.96 s, 792.47 s). Also, the standard deviation of MPPEngine (34.29 s) is lesser than jGenProg (153.86 s). On average, jGenProg takes up to 79.27 s to validate one program variant, whereas MPPEngine consumes only 33.70 s. We calculated this average value using executed test case count and the number of generations iterated for the bug. So, the time consumption on patch validation is reduced to 57.50%. This shows our approach is better in terms of reducing test execution time and cost, effectively.

Table 2. Time taken to validate a program variant by jGenProg and MPPEngine.

Time Cost	jGenProg	MPPEngine
Min	4.80 s	0.86 s
Median	30.96 s	19.53 s
Max	792.47 s	119.75 s
Std.DEV	153.86 s	34.29 s
Average	79.27 s	33.70 s
Time Reduction	57.50%	

6.2. RQ2: Does MPPEngine Produce More Patches for Bugs Compared to jGenProg?

In this research question, we analyzed the number of patches generated for each bug by both the approaches and compared the results to find which has better repair efficiency. Complete results of all the 27 bugs with patches are tabulated in Table 3.

As shown, the Bug column represents the name/ids of the bugs from the Defects4j dataset. Column 2 shows approach names, and column 3 is for the patches produced for each bug through both the approaches. The TC Count column shows the total count of executed test cases to produce the patches. The difference between the generated patch count of jGenProg and MPPEngine is available in column 5, and the approach that generated more patches is mentioned in the last column. Results displayed in Table 3 shows that, for few bugs, the MPPEngine executes more test cases than jGenProg in the given time. However, in most of such cases, MPPEngine produced more patches as it could validate more program variants by eliminating invalid patches earlier.

For example, in the given 3 h, Chart 3 executed 192,418 test cases through jGenProg and produced 9 patches where MPPEngine executed 1,387,003 test cases and produced 32 patches. Even though our approach takes more test cases, it produces more test suite adequate patches. Here, it has been proved that within the given time, MPPEngine validates more program variants and produces equal or more patches compared to jGenProg. As a result, MPPEngine produced patches for Chart 12 (1), Math 20 (34), Math 28 (142), Math 32 (6), Math 49 (26), and Math 50 (112) but no patch produced by

jGenProg. However, for Chart 14, jGenProg produces 7 patches where MPPEngine produced none in the given time.

Table 3. Experiment results of jGenProg and MPPEngine on Defects4j benchmark.

Bug	Approach	No. of Patches	TC Count	Patch Diff. Count	Better?
Chart 1	jGenProg	226	1,129,624	4	MPPEngine
	MPPEngine	230	1,158,113		
Chart 3	jGenProg	9	192,418	23	MPPEngine
	MPPEngine	32	1,387,003		
Chart 5	jGenProg	2	144,165	0	Same
	MPPEngine	2	649,313		
Chart 7	jGenProg	6	387,157	0	Same
	MPPEngine	6	379,745		
Chart 12	jGenProg	x	x	1	MPPEngine
	MPPEngine	1	21,118		
Chart 13	jGenProg	550	2,078,382	60	jGenProg
	MPPEngine	490	1,751,298		
Chart 14	jGenProg	7	182,591	7	jGenProg
	MPPEngine	x	459,574		
Chart 15	jGenProg	36	169,752	13	jGenProg
	MPPEngine	23	133,133		
Chart 25	jGenProg	51	1,073,187	28	MPPEngine
	MPPEngine	79	323,300		
Chart 26	jGenProg	38	1,682,156	11	MPPEngine
	MPPEngine	49	792,467		
Math 20	jGenProg	x	x	34	MPPEngine
	MPPEngine	34	67,160		
Math 28	jGenProg	x	x	142	MPPEngine
	MPPEngine	142	129,675		
Math 32	jGenProg	x	x	6	MPPEngine
	MPPEngine	6	16,351		
Math 49	jGenProg	x	x	26	MPPEngine
	MPPEngine	26	40,314		
Math 50	jGenProg	x	x	112	MPPEngine
	MPPEngine	112	121,654		
Math 60	jGenProg	2	27,794	0	Same
	MPPEngine	2	25,492		
Math 64	jGenProg	1	118,020	0	Same
	MPPEngine	1	118,304		
Math 70	jGenProg	2	153,420	0	Same
	MPPEngine	2	154,858		
Math 71	jGenProg	1	26,482	2	MPPEngine
	MPPEngine	3	36,320		
Math 73	jGenProg	17	112,824	0	Same
	MPPEngine	17	108,728		
Math 78	jGenProg	1	389,952	11	MPPEngine
	MPPEngine	12	121,657		

Table 3. Cont.

Bug	Approach	No. of Patches	TC Count	Patch Diff. Count	Better?
Math 80	jGenProg	39	105,032	0	Same
	MPPEngine	39	99,889		
Math 81	jGenProg	47	116,318	7	jGenProg
	MPPEngine	40	98,021		
Math 82	jGenProg	44	139,560	0	Same
	MPPEngine	44	130,337		
Math 84	jGenProg	1	15,928	0	Same
	MPPEngine	1	15,672		
Math 85	jGenProg	105	316,203	9	MPPEngine
	MPPEngine	114	254,046		
Math 95	jGenProg	17	50,598	0	Same
	MPPEngine	17	63,641		

Therefore, MPPEngine produced equal or more patches than jGenProg except for 4 bugs (14.82%): Chart 13, Chart 14, Chart 15, and Math 15 bugs for which jGenProg produced more patches. However, it is clear that the repair efficiency of MPPEngine is at least good as jGenProg for 10 bugs (37.04%) and better than jGenProg for 12 bugs (44.5%).

To prove the overall performance of our MPPEngine, we performed a statistical data analysis on both approaches in terms of the number of patches generated.

From Table 4, we can observe the sample statistics for the number of patches generated by jGenProg and MPPEngine. As per the statistics, there was a significant difference in the scores of jGenProg (Mean = 44.52, Standard Deviation = 111.43) and MPPEngine (Mean = 56.44, Standard Deviation = 102.02). This data analysis has been conducted on 27 observations (data) in total with p value = 0.05 and obtained t (27) = 1.65. The Mean value suggests that jGenProg does produce less patches compared to MPPEngine. Also, the standard error of MPPEngine (19.63) is lesser than jGenProg (21.45). This also means that the smaller the standard error, the less the spread and the more likely it is that any sample mean is close to the population mean. So, when a buggy program is processed through MPPEngine, the number of patches generated is increased. Therefore, it is evidence that, on average, our module MPPEngine does lead to improvements.

Table 4. Paired sample statistics for the number of patches generated by jGenProg and MPPEngine.

Pairs	Mean	N	Std. Deviation	Std. Error
No. of Patches by jGenProg	44.52	27	111.43	21.45
No. of Patches by MPPEngine	56.44	27	102.02	19.63

6.3. RQ3: Is There Any New Patch Generated for Bugs by MPPEngine for Which jGenProg Does Not Produce Any Solution?

In this section, we investigate whether our proposed approach produced any new patch to the buggy programs in the given time limit and within maximum generation. Figure 3 compares the patches generated by both jGenProg and MPPEngine.

There are patches generated for 27 bugs in which 20 bugs (20/27) were repaired in the same way by both approaches. While MPPEngine generates patches for 26 bugs out of 27 bugs except for Chart 14, whereas jGenProg generates patches for this bug (Chart 14). On the other hand, MPPEngine produces patches for the following 6 bugs: Chart 12 and Math 20, Math 28, Math 32, Math 49, and Math 50, which have not been repaired by jGenProg. We can witness the detailed results in Table 3. Let us consider the results of Math 28 bug, for which MPPEngine generated 142 test suite adequate

patches whereas, for the same bug, jGenProg executed 244,216 test cases, but no patch was generated in the given time. In another example, for Chart 1 bug, MPPEngine consumes slightly more test cases but produces four more test suite adequate patches than jGenProg. It proves that our fitness function filters out the best candidates for the re-modification so that our method produced more patches.

Figure 3. Number of patches generated per bug—jGenProg, MPPEngine.

In RQ3, in fact, we focus only on the bugs repaired by both the approaches in terms of repair effectiveness. It is clear that MPPEngine performs as good as jGenProg. However, in most cases, MPPEngine (26/27) outperforms jGenProg (21/27). As a result, the repair effectiveness of jGenProg is 77.8%, whereas MPPEngine is about 96.3% for the 27 bugs. Here, again, MPPEngine outperforms jGenProg.

7. Conclusions and Future Work

In this paper, we present the Modification Point Aware Test Prioritization and Sampling technique to reduce the time consumed by the patch validation of APR. MPTPS helps in detecting the invalid patches early in the validation process while reducing the size of test case execution. Meanwhile, in the remaining time, it validates more program variants that helps in enhancing repair effectiveness by finding more patches. Also, we introduced a new fitness function that improves repair efficiency. We built MPPEngine, including MPTPS and fitness function, by extending jGenProg in the Astor workspace.

We evaluated our approach MPPEngine on the Defects4j benchmark and compared our results against jGenProg, a state of the art tool for automated java programs. The repeated experiments on 48 Defects4j bugs from four different subjects on MPPEngine proved to reduce the average validation time for one program variant to 33.70 s, which is a 57.50% reduction compared to jGenProg, which takes 79.27 s. As for repair effectiveness, our MPPEngine generates patches in most cases (26/27) and outperforms jGenProg (21/27). The experimental results clearly depict that MPPEngine performed better in terms of repair efficiency as it uses a comparatively smaller number of test cases to validate the patches (and finds the invalid patches earlier), yet it provides better results.

Even though our proposed approach generates equal or more test adequate patches by eliminating invalid patches, there can still be some overfitting patches. Therefore, in the future, we plan to provide more test coverage by generating additional test cases using the Dynamic Symbolic Execution tool

to filter more incorrect patches. Although our proposed fitness function works fine, if we include additional test cases, it might need some refinement. So we need to analyze it thoroughly and improve it to provide more optimal results.

Supplementary Materials: The implementation and experiment results are available online at <https://github.com/yazhiniv/astor/tree/MPPEngine>.

Author Contributions: Conceptualization, Y.V.; methodology, Y.V.; validation, Y.V., P.Q.-N. and L.E.; formal analysis, Y.V., P.Q.-N.; investigation, Y.V., P.Q.-N. and L.E.; writing—original draft preparation, Y.V.; writing—review and editing, Y.V., L.E.; supervision, L.E.; project administration, L.E.; funding acquisition, L.E. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by the Next-Generation Information Computing Development Program (2017M3C4A7068179), and the Basic Science Research Program (2019R1A2C2006411) through the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIT).

Acknowledgments: I thank Martinez, M. et al. for their implementation of jGenprog in Astor. I also extend my thanks to my co-author Phung Quang-Ngoc and Lee Eunseok for their guidance throughout the implementation.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

APR	Automatic(ed) Program Repair
MPTPS	Modification Point aware Test Prioritization and Sampling
MPPTable	Modification Point aware Prioritized Table
MPPEngine	Modification Point aware Prioritized and sampled Engine

References

- O'Dell, D.H. The debugging mind-set. *Commun. ACM* **2017**, *60*. [CrossRef]
- Undo Software: Increasing Software Development Productivity with Reversible Debugging. 2014. Available online: https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf (accessed on 29 January 2020).
- Britton, T.; Jeng, L.; Carver, G.; Cheak, P.; Katzenellenbogen, T. *Reversible Debugging Software—Quantify the Time and Cost Saved Using Reversible Debuggers*; University of Cambridge, The Cambridge MBA: Cambridge, UK, 2013.
- Capgemini. World Quality Report. 2019. Available online: <https://www.capgemini.com/service/world-quality-report-2018-19/> (accessed on 29 January 2020).
- Murphy-Hill, E.; Zimmermann, T.; Bird, C.; Nagappan, N. The design of bug fixes. In Proceedings of the International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 332–341.
- Zeller, A. Automated debugging: Are we close. *Computer* **2001**, *34*, 26–31. [CrossRef]
- Gazzola, L.; Micucci, D.; Mariani, L. Automatic Software Repair: A Survey. *IEEE Trans. Softw. Eng.* **2019**, *45*, 34–67. [CrossRef]
- Elbaum, S.; Rothermel, G. Prioritizing Test Cases for Regression Testing Prioritizing Test Cases for Regression Testing. *IEEE Trans. Softw. Eng.* **2000**, *27*, 929–948.
- Jürgens, E.; Pagano, D.; Göb, A. *Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites*; Whitepaper; CQSE: Continuous Quality in Software Engineering GmbH: München, Germany, 2018.
- Le Goues, C.; Nguyen, T.V.; Forrest, S.; Weimer, W. Automatic Software Repair: A Survey. *IEEE Trans. Softw. Eng.* **2012**, *38*, 54–72. [CrossRef]
- Martinez, M.; Monperrus, M. Astor: Exploring the Design Space of Generate-and-Validate Program Repair beyond GenProg. *J. Syst. Softw.* **2019**, *151*, 65–80. [CrossRef]
- Martinez, M.; Monperrus, M. *ASTOR: Evolutionary Automatic Software Repair for Java*; Cornell University Library: Cornell, NY, USA, 2014.
- Martinez, M.; Monperrus, M. ASTOR: A program repair library for java. In Proceedings of the International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 441–444.

14. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–25 July 2014; pp. 437–440.
15. Qi, Y.; Mao, X.; Dai, Z.; Qi, Y. Efficient automatic program repair using function-based part-execution. In Proceedings of the International Conference on Software Engineering and Service Science, Beijing, China, 23–25 May 2013; pp. 235–238.
16. Yang, J.; Zhikhartsev, A.; Liu, Y.; Tan, L. Better test cases for better automated program repair. In Proceedings of the 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 831–841.
17. Khatibsyarbini, M.; Isa, M.A.; Jawawi, D.N.A.; Tumeng, R. Test case prioritization approaches in regression testing: A systematic literature review. *Inf. Softw. Technol.* **2018**, *93*, 74–93. [\[CrossRef\]](#)
18. Qi, Y.; Mao, X.; Lei, Y. Efficient automated program repair through fault-recorded testing prioritization. In Proceedings of the IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, 22–28 September 2013; pp. 180–189.
19. Qi, Y.; Mao, X.; Lei, Y.; Dai, Z.; Wang, C. The strength of random search on automated program repair. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 254–265.
20. Yoo, S.; Harman, M. Regression testing minimization, selection, and prioritization: A survey. *Softw. Test. Verif. Reliab.* **2012**, *22*, 67–120. [\[CrossRef\]](#)
21. Jang, Y.; Phung, Q.N.; Lee, E. Improving the Efficiency of Search-Based Auto Program Repair by Adequate Modification Point. In Proceedings of the 13th International Conference on Ubiquitous Information Management and Communication, Phuket, Thailand, 4–6 January 2019; pp. 694–710.
22. Fast, E.; Le Goues, C.; Forrest, S.; Weimer, W. Designing Better Fitness Functions for Automated Program Repair. In Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation—GECCO, Portland, OR, USA, 7–11 July 2010; pp. 965–972.
23. de Souza, E.F.; Goues, C.L.; Camilo-Junior, C.G. A novel fitness function for automated program repair based on source code checkpoints. In Proceedings of the GECCO '18, Kyoto, Japan, 15–19 July 2018; pp. 1443–1450.
24. Gzoltar Homepage. 2017. Available online: <http://www.gzoltar.com> (accessed on 15 August 2019).
25. Abreu, R.; Zoetewij, P.; van Gemund, A.J.C. On the accuracy of spectrum-based fault localization. In Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques, Windsor, UK, 12–14 September 2007; pp. 89–98.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).