IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 18, 2022, accepted March 11, 2022, date of publication April 4, 2022, date of current version May 17, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3164780

Improving Search-Based Automatic Program Repair With Neural Machine Translation

DONGCHENG LI[1], W. ERIC WONG[1], MINGYONG JIAN[2], YI GENG[2], AND MATTHEW CHAU[1]

[1]Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75080, USA
[2]School of Computer Science, China University of Geosciences, Wuhan 430074, China

Corresponding author: W. Eric Wong (ewong@utdallas.edu)

This work was supported by the USA National Science Foundation under Grant 1757828, Grant 1822137, and Grant 2050869.

ABSTRACT The challenge of automatically repairing bugs in programs to reduce debugging expenses and increase program quality is known as automated program repair. To overcome this issue, test-suite-based repair techniques use a specified test suite as an oracle and alter the input faulty program to pass the full test suite. GenProg is a well-known example of this kind of repair, in which genetic programming is used to reorder the statements already present in the faulty program. However, recent practical experiments suggest that GenProg's performance, notably for Java, is not sufficient. Improved program dependability necessitates the use of automatic program repair techniques. Template-based program repair techniques have recently been combined with search-based techniques to solve program issues automatically. Although intriguing, it has two fundamental drawbacks: Its search space often lacks the correct solution, and the technique disregards program expertise, such as precise code language. Compared with the template-based program repair approach, existing neural-machine-translation-based approaches are not limited by these constraints due to their ability to learn and generate new solutions. We propose an approach that combines a search-based automatic program repair technique with a neural-machine-translation-based approach. More specifically, we use both redundancy assumption and sequence-to-sequence learning of correct patches as the source for potential fix statements that feed into a multiobjective evolutionary search algorithm to find test-suite-adequate patches. In this work, a novel framework called ARJANMT is introduced for automatically repairing Java programs. Two sets of controlled experiments are conducted on 410 bugs from two benchmarks to investigate the repairability and correctness of our proposed framework. A comparison between state-of-the-art automatic program repair frameworks is made. The experimental results indicate that combining those two types of repair techniques (search-based and neural-machine-translation-based) produces better results or fixes bugs that they previously were unable to fix individually.

INDEX TERMS Search-based software engineering, automatic program Repair, multi-objective optimization, genetic programming, neural machine translation.

I. INTRODUCTION

Software bugs damage the quality of the product and seriously hinder the user experience [1], [2]. Manual debugging is complex and time-consuming. Automatic program repair (APR) is a technique to debug a faulty program automatically. It is the process of finding complete workable solutions to software bugs without the help of humans [3], making APR techniques quite useful and critical [4]. Search-based APR is based mainly on redundancy assumption and the evolutionary search algorithm to generate candidate patches. Neural machine translation (NMT), a commonly used technique for natural language processing (NLP) tasks, has recently been used to create adequate patches from buggy source code [4]–[6]. Due to redundancy assumptions, search-based APR tools such as GenProg and ARJA cannot build a patch when the repair ingredients are not present in the program's source code. This problem is considerably more prevalent in small projects. However, NMT models in bug-fixing tasks have a problem in precisely representing the context, and owing to the variety of errors and fixes, a single NMT model utilizing the ''optimal'' control parameter would

The associate editor coordinating the review of this manuscript and approving it for publication was Jian Guo.

struggle to generalize on the APR problem. Moreover, many NMT-based repairing approaches often repair single-location errors and can create only single-point repairs, not multilocation repairs. Additionally, NMT-based repair approaches such as SequenceR do not take into account operations such as insertion or deletion, which are generally taken into account by search-based APR approaches [4]. Consequently, the two techniques of search-based APR and NMT-based APR may be complementary.

Therefore, to improve the low recall rate, low accuracy, and low efficiency present in the existing APR techniques, an improved automated patch-generation tool for Java programs is proposed in this paper that is based on the multiobjective genetic programming and sequence-to-sequence learning for adequate patch-finding. NMT-based repair approaches are utilized to expand the search space of potential patches for search-based APR. Moreover, our framework is built on ARJA, and the sequence-to-sequence model that we employed in our research to enhance the search space of candidate patches is based on SequenceR. This paper makes the following contributions:

- We leverage both redundancy assumption and sequence-to-sequence learning for creating candidate fix statements that feed into a multiobjective evolutionary search method to identify test-suite-adequate patches.
- We design and implement a framework that combines the key concepts of ARJA and SequenceR into a unified repair framework, allowing testers to benefit from the capabilities of each component.
- We conduct a controlled experiment on 410 bugs from two benchmarks to investigate the repairability and correctness of our proposed framework. State-of-the-art APR frameworks are compared, and the benefits of combining search-based repair and NMT-based repair are also explored.

Moreover, this study is the first to combine the concepts of search-based repair and NMT-based repair into a unified repair framework, allowing testers to benefit from their respective component characteristics. The proposed APR approach is integrated into the RepairThemAll framework developed by [3], which automates and simplifies the execution of repair tools on various benchmarks.

The following sections of the paper are arranged as follows: Section 2 introduces the preliminaries of the study. Section 3 presents the proposed approach and tools for the automated repair of Java programs. The experimental design and result analysis are explained in Section 4 and 5. Finally, the paper is concluded, and directions for future research are provided in Section 6.

II. PRELIMINARIES

APR attempts to patch program errors based on a specification [7]. When a test suite is utilized as the specification, the paradigm is known as test-suite-based repair. At least one negative (i.e., first failing) test, as well as several positive (i.e., initially successful) tests, should be included in the test suite to define expected program behavior. A problem is deemed fixed or repaired in test-suite-based repair if a created patch allows the entire test suite to pass, which is called a test-adequate patch, also known as a plausible patch. Evolutionary computation-based repair is a popular category of test-suite-based repair techniques [8]. These methods discover a search space that may include accurate patches and then explore an optimization technique with a predefined fitness evaluation function [7].

A. SEARCH-BASED AUTOMATIC PROGRAM REPAIR

Search-based APR is a test-suite-based Generate and Validate APR approach. The foundations of search-based APR are (a) redundancy assumption, which entails that the components for a repair are already present somewhere in the current program and (b) the evolutionary search for a suitable patch, followed by generation and validation of the candidate patch [3]. Search-based APR approaches are crucial for enabling comprehensive automated diagnostic of software programs. They generate the program patch based on the provided faulty program and test suite to fix the program error [3]. Some recent state-of-the-art search-based APR techniques are discussed below.

1) GENPROG

GenProg [9] is a tool that automatically generates repairs for real-world bugs in off-the-shelf, legacy programs using existing unit tests. A repair is a patch that includes one or more code modifications that, when applied to software, allow it to pass a series of tests, which generally contain both functional passing tests and tests encapsulating bugs. During the use of GenProg, no explicit requirements, program annotations, or other coding practices are necessary. GenProg takes a program with errors and a set of test cases as input. It can be used on the entire program or just one class at a time. Using genetic programming, it searches for a program variation that accomplishes its tasks but is not susceptible to the defect. Genetic programming is a form of random search based on how organisms evolve over time [9]. It searches and generates modifications from an abstract syntax tree that may patch an error in the underlying program and creates new program variants using mutation and crossover. Each program variation is evaluated by a user-defined fitness function. Input test cases are used to evaluate the fitness, and individuals with high fitness are picked for further evolution. The process of GenProg ends when a variant passes all given test cases or the termination condition, such as time and iterations, is met [9]. However, as stated by [7], GenProg applies only genetic operators to high-granularity modifications, and its crossover does not generate additional edits. In addition, the main issue with genetic programming so far has been the amount of data to search through to identify the right program. This issue is also permanent in GenProg.

2) ARJA

ARJA [8] is a Java program repair system that employs multiobjective evolutionary search and a variety of methods

for reducing search space. It takes a faulty program and a JUnit test suite with at least one failed test as inputs. The remaining tests are passing tests. The programs corresponding to these tests are run as expected, and these tests are used as a baseline for specifying how the program should behave. A fault-localization method with the given input is used to seek the likely buggy statement for evolutionary search [8]. Coverage analysis is performed to maintain a record of every statement that each JUnit test visits. The source of the replacements or insertion ingredient statements for each likely buggy statement is selected from the seed statements generated by the coverage analysis, the current variable and method scope taking into consideration. The evolutionary search begins after the operations on likely buggy statement, such as delete, replace, and insert, are defined. More specifically, a multiobjective evolutionary search method is used to evolve the patches based on the representation of Genetic programming to keep the test failure rate and patch size as low as possible [8]. A rule-based search-space-reduction method with a set of criteria is used. A feasible fix is generated when a nondominated solution with a failure rate of 0 is obtained.

3) ARJA-E

ARJA-e [7] is an improved version of ARJA. It includes several new features and combines redundancy assumption, repair templates, lower-granularity patches, antipatterns, and patch size minimization into a single repair framework. ARJA-e searches for smaller patches using a non-dominated sorting genetic algorithm II (NSGA-II) within a reduced search space given by both redundancy assumption and repair templates. It employs the antipatterns to assign possible operation types for each potential buggy statement [7]. ARJA-e inputs a program with at least one failed test and a JUnit test suite. Fault localization, test-screening, redundant assumption, repair-template utilization, operation-type setting, and multiobjective evolutionary patch-generation are some of the primary stages involved in ARJA-e [7].

Automated generate-and-validate APR techniques typically rely on hard-coded rules, fixing only the errors that fit specific patch patterns [6]. Finding these rules requires much time and effort, and adapting them to different computer languages is difficult [6]. NMT-based APR approaches have been developed to address these challenges.

B. NEURAL-MACHINE-TRANSLATION-BASED APR

APR approaches based on neural machine translation (NMT) offer two significant benefits. First, NMT models learn intricate correlations between input and output processes that are difficult to deduce manually. NMT models may be able to reveal complicated links between buggy and clean codes that manual repair patterns cannot. Second, whereas actual generate and validate methods frequently use hard-coded fix templates that are specific to the scripting language, NMT-based repair methods can be reprogramed for a variety of languages and applications without having to create the fix patterns from scratch, which saves a significant length of time for developers. However, those approaches have two significant limitations [5]. Their search area frequently lacks a feasible solution, and their approach ignores program knowledge, such as precise code syntax. The results of applying NMT to program repair have been encouraging. A few recent NMT-based APR tools are discussed below.

1) SEQUENCER

SequenceR [4] is an APR framework that employs a sequence-to-sequence model that tries to fix bugs by making one-line fixes (i.e., the bug can be fixed by replacing a single buggy line with a single fixed line). The encoder and decoder with attention and copy mechanisms of the sequence-to-sequence network are recurrent neural networks (RNN) with long short-term memory (LSTM) gates [10], which are used to transform the abstract buggy context of a bug to the corresponding fixed code. Moreover, backpropagation is used to train the sequence-to-sequence model on a set of data.

2) COCONUT

CoCoNuT [6] is an APR framework that combines a novel context-aware NMT architecture and convolutional neural networks (CNNs). Using two separate encoders, each context-aware model records different information of the repair operation and their context, one for faulty lines and one for the context. Unlike SequenceR, CoCoNuT uses a CNN [11] instead of an RNN due to its capability of stacking layers for extracting hierarchical features and source code modeling for different statements and functions (multiple granularity levels). It can learn several repair strategies that can be used to correct a variety of faults while overcoming the limitations of conventional NMT methods by mixing these models [6]. It inputs a buggy statement and its context information. The data is tokenized before being supplied to the models. Each model then generates a list of patches for validation against the given test suite [6].

3) CURE

CURE [5] is an APR tool that uses subword tokenization to minimize the search space to increase the number of precise repairs. It also adapts an effective search technique to better identify and rank correct patches. Similar to SequenceR, CURE is also divided into three stages: training, inference, and validation. A large number of codes collected from open-source Java applications are used to pretrain the generative pretrained transformer (GPT) programming language model for CURE to learn human-written source code. Then, CURE fine-tunes the programming language model for the APR task. In addition, CURE develops a novel code-aware search approach that prioritizes plausible patches and patches close in length to the faulty code to find more correct fixes [5].

In this study, the benefits of combining both search-based repair and NMT-based repair are explored.

III. PROPOSED APPROACH

Existing APR approaches have a low recall rate, low accuracy, and low efficiency in generating plausible and

correct patches. Moreover, they often rely on hard-coded rules and fix only the faults that fit special patch patterns. To address these issues using EvoSuite and SequenceR as the base implementation, this paper presents an improved APR framework for Java programs that integrates the NMT technique into search-based APR. More specifically, it utilizes both redundancy assumption and sequence-to-sequence learning of correct patches as the source for possible fix statements that feed into a multiobjective evolutionary search algorithm to find test-suite-adequate patches. The benefit of combining the two APR approaches (search-based and NMT-based) is that the search-based patch generator will not be limited by the existing search space and hard-coded rules, and the trained sequence-to-sequence model will create possible repair ingredients and add to the candidate patch search space for the multiobjective evolutionary search to build test-suite adequate patches.

The following are the research questions addressed in this study:

RQ 1. Is it beneficial to exploit the search-based repair and NMT-based repair or combine them?
RQ 2. How effective is ARJANMT compared to state-of-the-art repair systems on benchmarks of bugs?
RQ 3. What is the overall correctness of the patches generated by ARJANMT compared to human-written patches, and other APR tools?
RQ 4. Does ARJANMT behave similarly across different benchmarks (Defect4J and Bears)?

To demonstrate the benefits of exploiting both redundancy assumption and sequence-to-sequence learning of bugs, we compared ARJA with SequenceR; Figure 1 shows the comparison results. In terms of test-suite-adequate patches, ARJA can fix five bugs that cannot be fixed by SequenceR, and SequenceR can fix six bugs that cannot be fixed by ARJA. Moreover, given that ARJA and SequenceR present suitable performance complementarity, we conclude that an APR approach that leverages the complementary strengths of the search-based technique and the NMT-based technique is worth exploring.

It is noteworthy that to answer **RQ1**, we tested those tools on only 47 bugs from the Defect4j benchmark in this analysis because SequenceR requires preprocessed data to function and SequenceR provides data for part of defects in Defect4j. However, this is not a major concern for our assessment, as the subjects of the bugs are still diverse, controlled, and from a well-known benchmark.

Figure 2 provides an overview of the proposed APR framework, ARJANMT. The right block represents the search-based APR module, and the left block represents the NMT module. ARJANMT inputs a buggy program and a set of JUnit tests to start. Its primary goal is to make changes to the program so that all tests pass. It goes through the following main steps. A fault-localization method with a given input is used to find the potentially faulty statements and scope of those statements, In the search-based APR module, coverage analysis is used to identify the source of the replacement and

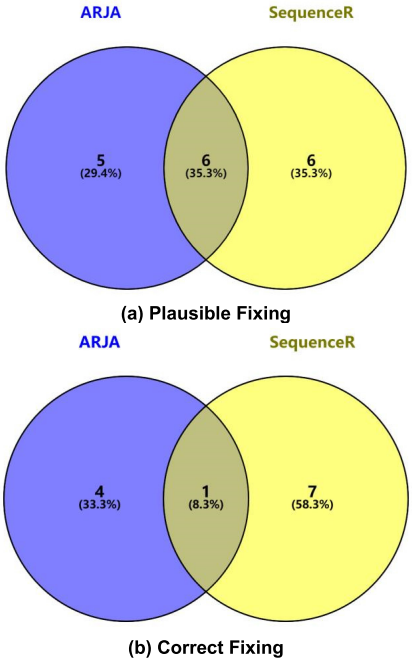

FIGURE 1. Venn diagram of repaired bugs: ARJA versus sequenceR.

insertion code from seed statements (statements that are visited by any JUnit test). Moreover, the potentially faulty lines and seed lines are parsed into the potentially faulty statements and seed statements respectively by the AST parser.

In the NMT module, the information of the bugs that can be fixed by replacing a single buggy line with a single fixed line is passed into the NMT module. The already trained NMT model on datasets contains buggy and repaired code then performs the buggy context abstraction. This process organizes the fault-localization data (e.g., buggy classes, methods, and lines) into a concise representation for the deep learning model but still has enough information about the bug's context to help predict how to fix it. The representation is then fed to a sequence-to-sequence model that has been trained to make the patch inference. Multiple single lines of code representing the bug's possible one-line patches can be generated by the patch inference. The generated code then enters the search-based APR module and becomes part of the potential fix ingredients, which are to be manipulated by the genetic programming.

The information obtained from both modules is passed into a genetic search algorithm with predefined operations, such as insertion, replacement, or deletion. The initial population can be seen as a collection of preliminary solutions of the search process. It is generated by mutated assertions with high suspicious values given by the fault localization. The crossover operator is then used to converge to globally optimal solutions for bug repair while quickly exploring the entire search space. To create candidate patches, we employed the learning model from SequenceR to provide additional fix ingredients for the multiobjective search-based repair strategy.

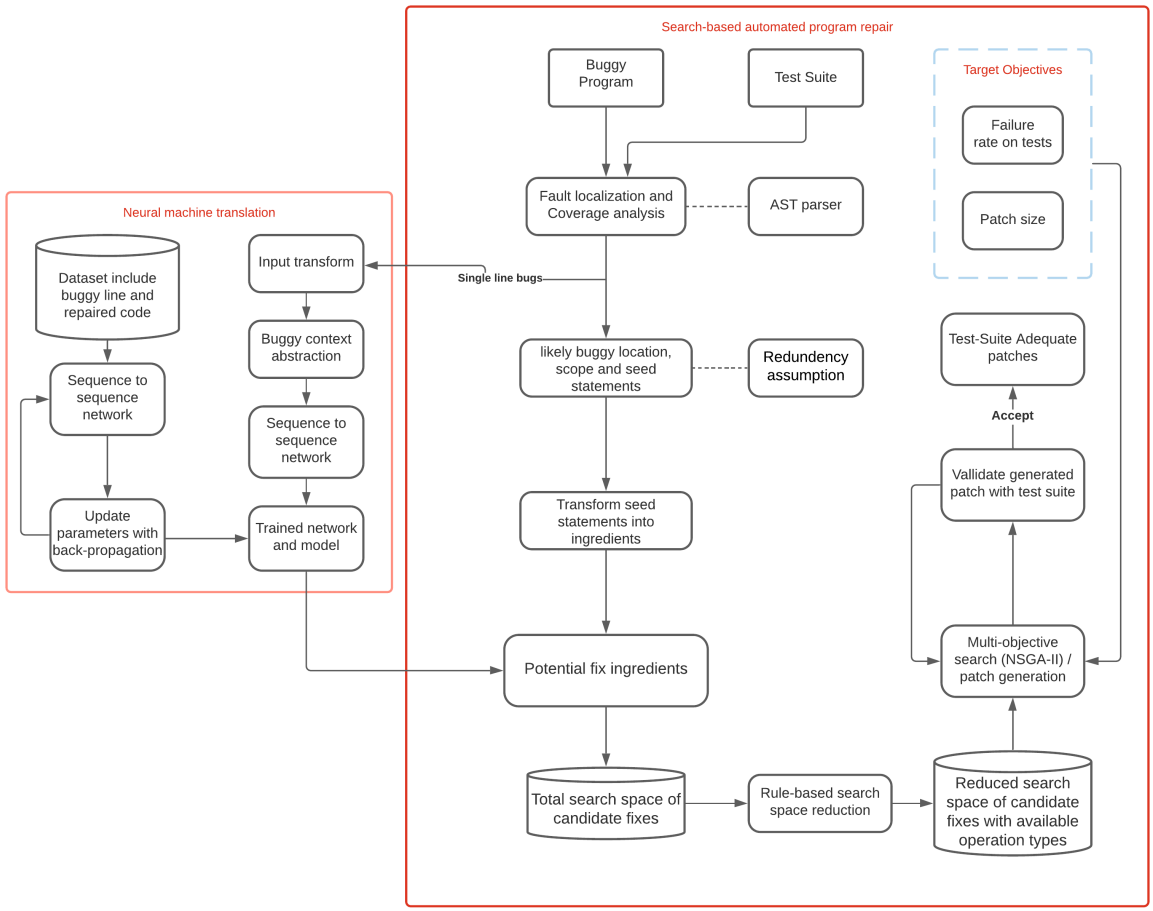

FIGURE 2. **Overview of the proposed automated program repair framework (ARJANMT).**

Algorithm 1 Invoke SequenceR

Input: Single-point fault-localization data produced by Ochiai
Output: Fault location with ingredient statements data
Initialize: Sequence-to-sequence model location L1, patch output location L2
Obtain source file location of fault L3 and fault line number N
Create a thread. Call SequenceR to run the shall script, passing through L1, L2, L3, N
Obtain all patch files generated by SequenceR through L2
getStatement method of SequencerPatchConvertor class is called to extract candidate component statements (ingredients) from all patch files
Add the extracted candidate ingredients to the candidate ingredients set of the faulty point
End

Algorithm 2 SequencerPatchConvertor

Input: Patch file directory, fault line number
Output: Candidate ingredients (statements) set
Input: Obtain all patch files in the directory.
The number is M
for i = 1 to M do
Select a patch file in the set
Generates an abstract syntax tree for the patch file
$S = \{s_1, s_2, \ldots, s_n\}$
for j = 1 to n do
Obtain the node of the abstract syntax tree s_j
if the start line number of $s_j s_j$ is the same as the fault line number **then**
Store node s_j into the candidate patch set
End If
End For
End For

Algorithm 1 shows the procedure of invoking SequenceR. It takes a single fault-localization data produced by Ochiai and outputs the candidate ingredients set of the faulty point.

Algorithm 2 demonstrates the procedure for converting SequenceR's unverified patches to potential fix ingredients used by multiobjective evolutionary search to find test-suite-adequate patches. The inputs are the patch file directory and the bug line number, and the output is a set of sentences.

IV. EXPERIMENT DESIGN

The experimental evaluation included a selection of benchmarks of bugs, as well as APR tools, and the bugs were repaired with these tools. The data was then gathered by running the selected repair tools on selected bugs, and the results were compared between the repair tools. The tool proposed in this paper is called ARJANMT. We performed the test on the following three high-performance workstations to ensure the results' fairness, trustworthiness, and dependability:

- Workstation 1 - Operating system: Linux Ubuntu, CPU: Intel i9, GPU: RTX 3090, RAM: 64GB
- Workstation 2 - Operating system: Linux Ubuntu, CPU: Intel i9, GPU: RTX 3070, RAM: 64GB
- Workstation 3 - Operating system: Linux Ubuntu, CPU: Intel i9, GPU: RTX 3070, RAM: 64GB

There were a total of 410 bugs (buggy programs) included in our experiment, including the following categories of bugs from Defects4J and Bears: chart (26 bugs), math (106 bugs), time (27 bugs), traccar-traccar (42 bugs), spring-projects-spring-data-commons (15 bugs), opentracing-contrib-java-p (1 bug), julianps-modelmapper-module-vavr (1 bug), dungba88-libra (1 bug), danfickle-openhtmltopdf (1 bug), apache-incubator-dubbo (5 bugs), albfernandez-GDS-PMD-Security-Rules (1 bug), SzFMV2018-Tavasz-AutomatedCar (2 bugs), 2018swecapstone-h2ms (3 bugs), Activiti-Activiti (1 bug), Activiti-activiti-cloud-app-service (1 bug), AxonFramework-AxonFramework (1 bug), Blazebit-blaze-persistence (1 bug), CSU-CS414-WareWolves-cs414-f18-001-WareWolves (1 bug), CorfuDB-CorfuDB (1 bug), DataBiosphere-consent-ontology (1 bug), DmitriiSerikov-money-transfer-service (1 bug), EnMasseProject-enmasse (3 bugs), FasterXML-jackson-databind (26 bugs), FasterXML-jackson-dataformats-binary (1 bug), FasterXML-jackson-dataformats-text (1 bug), HubSpot-Baragon (1 bug), INRIA-spoon (62 bugs), OpenFeign-feign (1 bug), SonarOpenCommunity-sonar-cxx (1 bug), SpoonLabs-gumtree-spoon-ast-diff (1 bug), aicis-fresco (5 bugs), apache-incubator-servicecomb-java-chassis (2 bugs), apache-incubator-tamaya (1 bug), apache-jackrabbit-oak (1 bug), apereo-java-cas-client (1 bug), aws-aws-encryption-sdk-java (1 bug), awslabs-amazon-kinesis-client (1 bug), brettwooldridge-HikariCP (1 bug), classgraph-classgraph (1 bug), cpesch-RouteConverter (2 bugs), ctripcorp-apollo (1 bug), dhis2-dhis2-core (1 bug), hexagonframework-spring-data-ebean (1 bug), javadev-underscore-java (1 bug), jenkinsci-ansicolor-plugin (1 bug), jgrapht-jgrapht (1 bug), kmehrunes-valuestreams (1 bug), lettuce-io-lettuce-core (1 bug), linkedin-pinot (1 bug), milaboratory-milib (1 bug), molgenis-molgenis (6 bugs), openmrs-openmrs-module-htmlformentry (1 bug), openmrs-openmrs-module-webservices.rest (1 bug), openzipkin-zipkin (1 bug), org-tigris-jsapar-jsapar (1 bug), paritytrading-foundation (1 bug), pippo-java-pippo (1 bug), rafonsecad-cash-count (2 bugs), raphw-byte-buddy (5 bugs), rkonovalov-jsonignore (1 bug), shapesecurity-shift-java (1 bug), smallcreep-cucumber-seeds (1 bug), societe-generale-ci-droid-tasks-consumer (1 bug), spring-cloud-spring-cloud-gcp (4 bugs), spring-projects-spring-data-jpa (2 bugs), square-javapoet (1 bug), swagger-api-swagger-codegen (2 bugs), thelastpickle-cassandra-reaper (3 bugs), thelinmichael-spotify-web-api-java (1 bug), vert-x3-vertx-jdbc-client (1 bug), vert-x3-vertx-web (1 bug), vitorenesduarte-VCD-java-client (1 bug), vkostyukov-la4j (1 bug), webfirmframework-wff (1 bug), debezium-debezium (7 bugs).

We ran each of the tools three times on each of the benchmark bugs, each time on a different workstation. The issue of whether the patches produced by a tool are correct beyond passing the test suite has been a heated topic of discussion. In our experiments, we manually checked and verified the correctness of the patches produced by different tools. Moreover, each produced patch was verified and validated by three different human programmers to eliminate single-person bias. We considered a fault to be correctly fixed if at least one of the generated patches from each of the three runs of a certain APR tool was recognized as correct. Hence, all generated patches were reviewed for correctness. The results of the trials are displayed in the table and figures below.

This research required about 5,000 hours of testing time and human labor. Our research was not concerned with the computational cost of running the ARJANMT tool; instead, we were primarily concerned with the repairability and accuracy of the patches created by our proposed approach. We integrated our automatic repair tool into a framework called RepairThemAll to test the effectiveness of our approach. In RepairThemAll, there are five open-source bug benchmarks; we employed two benchmarks in our experiment to lessen the bias that a single assessment might generate. We picked Defect4j and Bears because of their complexity (they contain the most lines of code) and their reputation in the APR community (they were often used in measuring the efficacy of APR tools). It is noteworthy that while Defects4J encompasses six projects, we did not take into account projects Lang, Mockito, and Closure because these projects frequently require more specialized tools to repair, and they frequently produce poor results for generalized APR tools. More explicitly, project Lang was dropped because it frequently encounters timeout issues. Project Closure was removed because it uses the specialized testing format rather than the typical JUnit tests. Mockito was ignored because it is a relatively new project added to the Defects4J benchmark [8].

To examine the effectiveness of our approach, we compared our approach, ARJANMT, to the three top-performing test-suite-based APR tools presented in the study by [3]: Nopol, DynaMoth, and ARJA. These three tools generate test-suite-adequate patches for the highest number of bugs. Nopol [12] is a semantics-based tool for repairing Java programs. It employs random values that make all failed test cases from the program under repair pass. It gathers these values while the program runs and incorporates them

into a satisfiability modulo theory (SMT) formula to obtain an expression that fits the behavior of the abovementioned random value. Finally, Nopol converts the SMT resolution into a source code patch when the SMT formula is satisfied [3]. Similar to Nopol, DynaMoth [13] also focuses on buggy and missing ''if'' conditions. However, instead of generating patches using SMT, it combines variable and method calls collected by Java Debug Interface to generate more complex fixes [3].

V. RESULT ANALYSIS

Figure 3 shows the repairability of the repair tools in descending order by the number of generated test-suite-adequate patches. It displays the number of unique bugs patched by the tool in blue and the number of patched bugs that were also patched by other repair tools in grey. The overall number of patched bugs with the percentage over 410 bugs for each tool is also presented.

ARJANMT generates patches for 32 bugs in total, that is, 12.2% of all bugs, four of which are patched solely by ARJANMT and 46 of which are patched by ARJANMT and other tools. ARJA synthesizes patches for 47 bugs in total, that is 11.5% of all bugs, one of which is patched exclusively by ARJA and 46 of which are patched by ARJA and other tools. Nopol patches 32 bugs in total, that is 7.8% of all bugs, 15 of which are patched only by Nopol and 17 of which are patched by Nopol and other tools. DynaMoth generates patches for 24 bugs in total, that is 5.9% of all bugs, four of which are fixed solely by DynaMoth and 20 of which are fixed by DynaMoth and other tools.

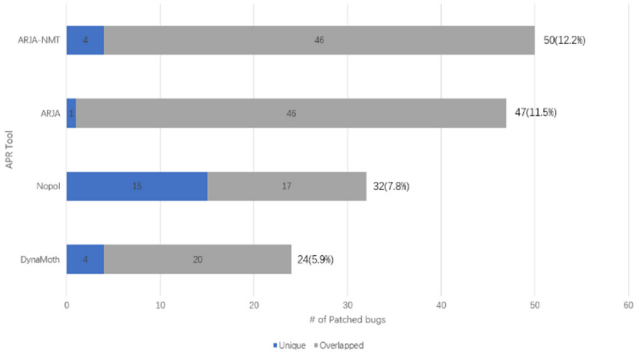

FIGURE 3. Repairability of the four repair tools on 358 bugs.

In terms of the number of patched bugs, we observe that ARJANMT is the tool that patches the highest number of overall bugs, and it is followed by ARJA, Nopol, and DynaMoth. However, Nopol is the tool that patches the highest number of unique bugs, and it is followed by ARJANMT, DynaMoth, and ARJA.

Table 1 shows the number of patched bugs overlapping among each pair of repair tools. The number of bugs uniquely patched by the tool is shown when the column name and line name are the same (main diagonal). ARJANMT, for example, has patched four bugs that are not patched by other tools, and

TABLE 1. The number of overlapped patched bugs per repair Tool.

	ARJANMT	Nopol	DynaMoth	ARJA
ARJANMT	4(8%)	14(28%)	16(32%)	45(90%)
Nopol	14(44%)	15(47%)	15(47%)	14(44%)
DynaMoth	16(67%)	15(63%)	4(17%)	15(63%)
ARJA	45(96%)	14(30%)	15(32%)	1(2%)

14 bugs that are patched by ARJANMT are also patched by Nopol. ARJA has a 90% overlap with ARJANMT but only a 44%–63% overlap with the other tools. In contrast, Nopol has a 63% overlap with DynaMoth but only a 28%–30% overlap with the other tools. These results indicate that the repair tools that are implemented in similar patch-generating frameworks exhibit significant overlap.

The term ''plausible fixing'' or ''test-adequate fixing'' refers to a tool's ability to generate a plausible or test-adequate patch for the given program that passes all unit tests. ''Correct fixing'' means that such a generated plausible patch must be semantically equal or identical to the human-written patch.

The Venn diagrams in Figure 4 answer **RQ2**. They depict the intersection of patched bugs. Figure 4(a) shows the test-adequate fixings while Figure 4(b) shows the correct fixings among ARJANMT, Nopol, ARJA, and DynaMoth. The vast majority of bugs that ARJA successfully fixes can also be fixed by ARJANMT. ARJANMT absorbs basic principles from ARJA, implying that the inclusion is quite successful, allowing ARJANMT to inherit almost all of ARJA's repairing powers.

Compared to ARJA and DynaMoth, Nopol shows better complementarity to ARJANMT in terms of test-adequate patches. For example, Nopol can patch 22 bugs that cannot be patched by ARJANMT. However, in terms of correct fixing, neither Nopol nor DynaMoth exhibits a complementarity effect to ARJANMT. Thus, we conclude that simply combining semantics-based Nopol with ARJANMT might not further enhance the performance of ARJANMT.

To answer **RQ3**and**RQ4**, Table 2 shows the repairability of ARJANMT, Nopol, DynaMoth, and ARJA and declares a patch correct if it is semantically equal or identical to the human-written patch. It is worth noting that Table 2 shows only the projects that have at least one bug patched by at least one APR tool. As shown in Table 2, in terms of the total number of patched bugs, ARJANMT surpasses all other tools. In particular, ARJANMT can generate test-adequate patches for 6.4% more bugs than ARJA can, 56.3% more than Nopol can, and 108.3% more than DynaMoth can. In terms of the bugs correctly patched, ARJANMT and ARJA have the same number of correct patches, higher than those of Nopol and DynaMoth. Moreover, as shown in the table, four repair tools all overfit Defects4J; that is, the overall number

TABLE 2. Comparison in terms of the number of bugs fixed and correctly fixed (plausible or correct).

		tools			
benchmark	project	ARJANMT	Nopol	DynaMoth	ARJA
Defects4J	chart	10/4	8/1	8/1	10/4
	math	24/11	23/1	16/1	23/12
	time	4/1	1/0	0/0	2/0
Bears	traccar-traccar	8/0	0/0	0/0	8/0
	julianps-modelmapper-module-vavr	1/0	0/0	0/0	1/0
	dungba88-libra	1/0	0/0	0/0	1/0
	albfernandez-GDS-PMD-Security-Rules	1/0	0/0	0/0	1/0
	SzFMV2018-Tavasz-AutomatedCar	1/0	0/0	0/0	1/0
Total		50/16	32/2	24/2	47/16

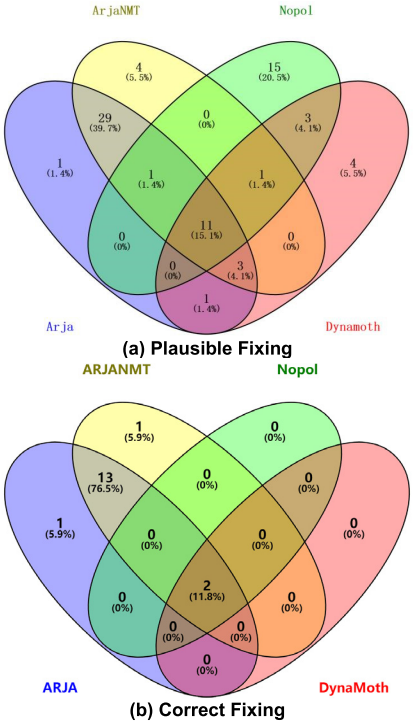

FIGURE 4. Venn diagram of repaired bugs.

of patches generated by all tools is much higher for bugs in Defects4J compared to Bears. However, the repairability of search-based APRs still outperforms that of semantics-based Nopol and DynaMoth for bugs in Bears.

VI. CONCLUSION

Automated program repair is the desired approach for software development. In this paper, we designed and implemented an automated program repair framework called ARJANMT, which utilizes both redundancy assumption and sequence-to-sequence learning of correct patches as the ingredients for the possible fix statements that feed into an NSGA-II algorithm to find test-suite-adequate patches. The results of the experiment indicate that it is beneficial to combine both the search-based repair technique and the NMT-based repair technique due to their complementary effects and that ARJANMT surpasses ARJA, Nopol, and DynaMoth by 6.4%, 56.3%, and 108.3% respectively in terms of the total number of patched bugs. In terms of bugs correctly patched by tools, ARJANMT and ARJA have the same, highest number of correct patches compared to Nopol and DynaMoth. Moreover, search-based APR has the potential to overcome the overfitting issue exhibited in the new APR benchmarks. The directions for future research can be threefold. The searchability of the multiobjective evolutionary algorithms and the repairability of the NMT model can be improved. In addition, template-based APR techniques can also be combined with ARJANMT to improve the performance of the tool.

REFERENCES

[1] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok, "Recent catastrophic accidents: Investigating how software was responsible," in *Proc. 4th Int. Conf. Secure Softw. Integr. Rel. Improvement*, 2010, pp. 14–22.
[2] W. E. Wong, X. Li, and P. A. Laplante, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *J. Syst. Softw.*, vol. 133, pp. 68–94, Nov. 2017.
[3] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 302–313.
[4] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SEQUENCER: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1943–1959, Sep. 2021.
[5] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1161–1173.
[6] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 101–114.
[7] Y. Yuan and W. Banzhaf, "A hybrid evolutionary system for automatic software repair," in *Proc. Genetic Evol. Comput. Conf.*, Jul. 2019, pp. 1417–1425.
[8] Y. Yuan and W. Banzhaf, "ARJA: Automated repair of Java programs via multi-objective genetic programming," *IEEE Trans. Softw. Eng.*, vol. 46, no. 10, pp. 1040–1067, Oct. 2018.
[9] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.

[10] R. A. Sulthana and A. J. A. K. Jovith, "LSTM and RNN to predict COVID cases: Lethality's and tests in GCC nations and India," *Int. J. Performability Eng.*, vol. 17, no. 3, pp. 299–306, Mar. 2021.
[11] D. Bhavana, K. K. Kumar, M. B. Chandra, P. V. S. K. Bhargav, D. J. Sanjana, and G. M. Gopi, "Hand sign recognition using CNN," *Int. J. Performability Eng.*, vol. 17, no. 3, pp. 314–321, Mar. 2021.
[12] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in Java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.
[13] T. Durieux and M. Monperrus, "DynaMoth: Dynamic code synthesis for automatic program repair," in *Proc. 11th Int. Workshop Autom. Softw. Test (AST)*, 2016, pp. 85–91.

DONGCHENG LI received the B.S. degree in computer science from the University of Illinois, Springfield, and the M.S. degree in software engineering from The University of Texas at Dallas, where he is currently pursuing the Ph.D. degree. His research interests include search-based software engineering and intelligent optimization algorithms.

W. ERIC WONG received the M.S. and Ph.D. degrees in computer science from Purdue University. He was at Telcordia Technologies (formerly, Bellcore) as a Senior Research Scientist and the Project Manager, where he was in charge of dependable telecom software development. He is currently a Full Professor and the Founding Director of the Advanced Research Center for Software Testing and Quality Assurance in Computer Science Department at The University of Texas at Dallas. He also has an appointment as a Guest Researcher with the National Institute of Standards and Technology (NIST), an agency of the U.S. Department of Commerce. He has very strong experience developing real-life industry applications of his research results. His research interest includes helping practitioners improve the quality of software while reducing the cost of production. In particular, he is working on software testing, debugging, risk analysis/metrics, safety, and reliability. In 2014, he was named the IEEE Reliability Society Engineer of the Year. He is the Editor-in-Chief of IEEE TRANSACTIONS ON RELIABILITY. He is also the Founding Steering Committee Chair of the IEEE International Conference on Software Quality, Reliability, and Security (QRS).

MINGYONG JIAN received the bachelor's degree in network engineering from the China University of Geosciences, Wuhan, where he is currently pursuing the master's degree in computer technology. His main research interests include intelligent dispatching and information engineering.

YI GENG received the bachelor's degree in the internet of things engineering from the Zhejiang University of Technology. He is currently pursuing the master's degree in computer technology from the China University of Geosciences, Wuhan. His main research interests include intelligent dispatching and information engineering.

MATTHEW CHAU is currently pursuing the undergraduate degree in electrical engineering with the University of Texas at Dallas. His research interests include software fault prevention, software reliability, and preventive engineering.