ROSE: An IDE-Based Interactive Repair Framework for Debugging

STEVEN P. REISS, Department of Computer Science, Brown University, USA
XUAN WEI, School of Computer Science, Wuhan University, China
JIAHAO YUAN, School of Computer Science, Wuhan University, China
QI XIN*, School of Computer Science, Wuhan University; Hubei Luojia Laboratory, China

Debugging is costly. Automated program repair (APR) holds the promise of reducing its cost by automatically fixing errors. However, current techniques are not easily applicable in a realistic debugging scenario because they assume a high-quality test suite and frequent program re-execution, have low repair efficiency, and only handle a limited set of errors. To improve the practicality of APR for debugging, we propose ROSE, an interactive repair framework that is able to suggest quick and effective repairs of semantic errors while debugging in an Integrated Development Environment (IDE). ROSE allows an easy integration of existing APR patch generators and can do program repair without assuming the existence of a test suite and without requiring program re-execution. It works in conjunction with an IDE debugger and assumes a debugger stopping point where a problem symptom is observed. ROSE asks the developer to quickly describe the symptom. Then it uses the stopping point, the identified symptom, and the current environment to identify potentially faulty lines, uses a variety of APR techniques to suggest repairs at those lines, and validates those repairs without re-executing the program. Finally, it presents the results so the developer can examine, select, and make the appropriate repair. ROSE uses novel approaches to achieve effective fault localization and patch validation without a test suite or program re-execution. For fault localization, ROSE builds on a fast abstract-interpretation-based flow analysis to compute a static backward slice approximating the real dynamic slice while taking into account the symptom and the current execution. For patch validation without re-running the program, ROSE generates simulated traces based on a live-programming system for both the original and repaired executions and compares the traces with respect to the problem symptoms to infer patch correctness. We implemented a prototype of ROSE that works in an Eclipse-based IDE and evaluated its potency and utility with an effectiveness study and a user study. We found that ROSE's fault localization and validation are highly effective and a ROSE-based tool using existing APR patch generators generated correct repair suggestions for many errors in only seconds. Moreover, the user study demonstrated that ROSE was helpful for debugging and developers liked to use it.

CCS Concepts: • **Software and its engineering → Software creation and management**; *Software development techniques.*

Additional Key Words and Phrases: Debugging, Interactive Repair Framework, Automated Program Repair, Integrated Development Environment

*Corresponding author.

Authors' Contact Information: Steven P. Reiss, Department of Computer Science, Brown University, USA, spr@cs.brown.edu; Xuan Wei, School of Computer Science, Wuhan University, China, isabel1015@whu.edu.cn; Jiahao Yuan, School of Computer Science, Wuhan University, China, yuanjiahao@whu.edu.cn; Qi Xin, School of Computer Science, Wuhan University; Hubei Luojia Laboratory, China, qxin@whu.edu.cn.

ACM 1557-7392/2024/11-ART111
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Steven P. Reiss, Xuan Wei, Jiahao Yuan, and Qi Xin. 2024. ROSE: An IDE-Based Interactive Repair Framework for Debugging. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (November 2024), 39 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Debugging is laborious. Automated program repair [29, 83, 144] (APR) aims to alleviate a developer's burden by automatically correcting program errors by identifying the failure-responsible locations (fault localization), generating patches for the locations (patch generation), and validating the patches (patch validation). Because people routinely use Integrated Development Environments (IDEs) to write code and debug, we envision integrating APR into an IDE to provide practical and useful debugging support. In order for this to occur, APR needs to work in conjunction with the IDE debugger to let the developer fix errors as they find them. APR needs to work without test cases since the environment cannot guarantee that a significant number of tests are available [8, 53, 55] or that the bug occurs while running a test case. APR needs to work without rerunning the program since errors can be found in situations where this is difficult (e.g., a long run or an interactive session). Finally, APR needs to be fast enough so that it can be used interactively [89].

Current techniques do not satisfy these requirements. Most techniques are test-based [29, 84, 144]. They assume the existence of an extensive test suite serving as the correctness criterion and rely on program re-execution against the test suite both to identify the locations to repair and to validate the patches generated. In practice, high-quality test suites are often unavailable. As suggested by [8, 53], developers often do not write a test suite containing a sufficient number of test cases or even do not write tests at all (especially for the initial development). Koyuncu et al. [55] noted that bugs are often reported without an available test suite being able to reveal them. For over 90% of the real bugs in the Defects4J dataset [45], the test cases relevant to the defective behavior of the software represent future data and were only introduced after the bug was identified. We also scanned all Java projects in the Merobase source repository [36] and found that, of the 12,525 projects with 50 or more methods, about 58% of them have no tests at all. In fact, only about 13% projects had as many tests per method as the packages in Defects4J used for evaluating APR techniques.

Even when a high-quality test suite is available, a test-based technique can still fall short for practical debugging, as its over-reliance on dynamic analysis for fault localization and patch validation can lead to decreased efficiency. As reported in the evaluation of some of the latest test-based techniques (for example CURE [42]), the average time taken to repair an error can be longer than 20 minutes[1]. A long repair time is not aligned with a user's expectation [89], especially when using an IDE.

Another factor that impedes the adoption of test-based techniques for debugging is need for program re-execution. In a realistic debugging scenario, recreating the environment for the immediate failure caused by the error can be difficult [44, 48, 99, 106] since the failure can be identified in a long run or in an interactive session.

Some of the recent techniques [6, 7, 25, 55, 110] have sought to achieve error repair without a test suite. These techniques, although promising, are still not fast [25] or are only applicable to specific types of errors such as the security vulnerabilities [25, 33] and heap-property faults [110], potential issues detected by static analyzers [6, 7] (not errors actually observed while debugging), and bugs that arise from special forms of specifications (e.g., bug reports [55]) or that require additional resources (e.g., historical patches [6]). Most recently, the Large Language Models (LLMs) have

[1]Most recent techniques were evaluated in a setting where a five-hour timeout is used for repair. The repair time used to find a correct patch was often not reported.

shown superior abilities in suggesting effective patches for error correction. LLM-based techniques are fast and do not appear to be tied to specific errors. However, they assume that the faulty code can be effectively pinpointed and do not contain a (non-test-based) patch validation component. As such, they are actually patch generators rather than full repair tools.

To sum up, current APR techniques are not well-suited for use while debugging in an IDE because they make strong and unrealistic repair assumptions, do not have the necessary efficiency, and often have a limited repair scope.

To improve the practicality of APR for debugging, we propose ROSE, an IDE-based framework that supports **R**epairing **O**vert **S**emantic **E**rrors. A semantic error can cause program runtime failures, which the developer tackles for debugging [143]. ROSE is not a repair technique but a framework that facilitates the integration of APR patch generators into an IDE to leverage the power of existing patch generation approaches to achieve effective debugging. ROSE aims for interactive and practical semantic error repair. It does not assume the existence of a test suite and does not require re-executing the program while debugging. With no test suite exposing the failure, ROSE starts by interacting with the developer to obtain a quick description of the problem symptom showing what is wrong at a break point. Based on this description, ROSE follows a generate-and-validate procedure to make repair suggestions by first performing fault localization to identify potential repair locations responsible for the problem, then invoking the integrated patch generators to generate potential repairs, and finally validating and sorting those repairs.

The key challenges are how to perform quick and effective fault localization and patch validation without a test suite and without program re-execution. ROSE uses a novel fault localization approach that starts with a fast abstract-interpretation-based flow analysis that is already available in an IDE, and uses the results of this analysis along with the problem symptom, current values from the debugger, and current run time stack to efficiently compute a limited backward slice starting from the location where the symptom is observed. The abstract interpretation associates an abstract program state including stack, variable, and memory values with each execution point. ROSE uses this information by working backwards one step at a time from the stopping point where the problem was observed, keeping track of which variables and stack elements are relevant at each point, and building a graph of control and data dependencies relevant to the symptom. The graph is restricted using current program values and the current execution stack, by limiting the number of times a variable is changed while being considered, and by limiting the number of control flow dependencies that are considered. ROSE then converts this graph into the set of lines that might affect the symptom.

Without using test cases or allowing dynamic program re-execution, patch validation is also challenging. ROSE tackles the problem by generating simulated traces that reflect the real executions of the failure-related portions of original and repaired programs and comparing the traces to infer patch correctness. The simulated traces are generated by a continuous-execution plugin for the IDE. Specifically, ROSE first obtains a baseline simulated execution that duplicates the original problem from some point on the current call stack. Then for each patch it obtains a repaired simulated execution from the same starting point. The trace comparison considers the specified symptom along with a variety of matching conditions to compute a score approximating the likelihood of the error being correctly repaired. Finally, ROSE presents the repairs as they are found in priority order. The developer can choose a repair to preview or have ROSE make the repair.

To assess the effectiveness and usefulness of ROSE, we implemented a prototype of the framework in the Eclipse-based Code Bubbles IDE [10], and conducted two studies. The first is an effectiveness study used to evaluate whether ROSE's approaches work. In this study, we implemented two ROSE-based repair tools, ROSE-PC and ROSE-PS. The former uses a combination of predefined patterns and ChatGPT (v3.5-turbo) [15] for repair generation and the latter is based on patch generators

using the patterns and a local-search-based approach [128] to produce repairs. We applied the tools to two published error benchmarks, QuixBugs [64] and a subset of Defects4J [45] for repair. We found that ROSE's core approaches, fault localization and patch validation, are very effective. The fault localization included the correct repair location for 89% of the errors, and the patch validation gave a top-5 rank for all correct repairs and was never a reason for ROSE's failure. Moreover, the ROSE tools quickly and correctly repaired many errors. The average time of finding a correct repair is only a few (less than 10) seconds. And the ROSE-PC tool, which uses advanced patch generators, found correct repairs for as many as 36/40 QuixBugs and 37/60 Defects4J errors.

The second study is a user study designed to evaluate the usability of ROSE. We recruited 26 participants to debug four programs either with or without a ROSE-based tool. Our results showed that ROSE helped 44% more participants succeed in a debugging task and helped reduce the debugging time by about 16.5%. Moreover, the feedback given by the participants reveals that they think ROSE is useful and they like ROSE. Overall, we believe that our results are encouraging, as they showed that the ROSE framework can empirically integrate existing APR patch generators into an IDE to provide quick and effective repair suggestions and help developers debug.

The main contributions of this work are:

- A working quick-repair framework that can integrate a variety of repair-generating mechanisms into an IDE to provide accurate suggestions in a reasonable time frame.
- Novel approaches for interactive problem specification, fault localization, and repair validation that do not require a test suite or program re-execution.
- Evaluations based on two studies that demonstrate ROSE's effectiveness and usefulness.
- A prototype of ROSE along with our experiment data and study result, publicly available at https://github.com/rose-apr/rose.

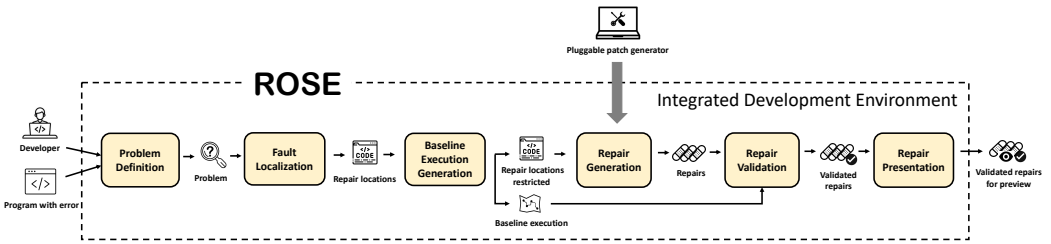
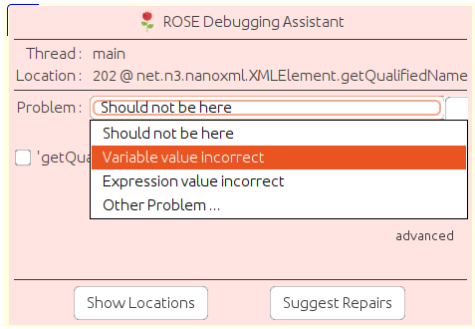


Fig. 1. Overview of ROSE.

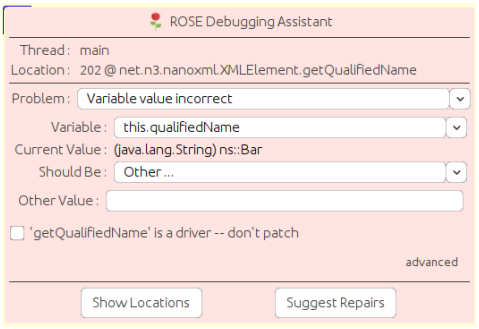
2 Overview of ROSE

ROSE is an interactive repair framework designed to work in conjunction with an IDE and a debugger. It performs six steps: *problem definition*, *fault localization*, *baseline execution generation*, *repair generation*, *repair validation*, and *repair presentation* to generate repair suggestions without a test suite. Figure 1 shows the workflow of ROSE. Figure 2 presents an example of the interactive interface of ROSE for problem specification (the top row), fault localization and repair suggestions (the middle row), and repair preview (the bottom row), which highlights the code changes.

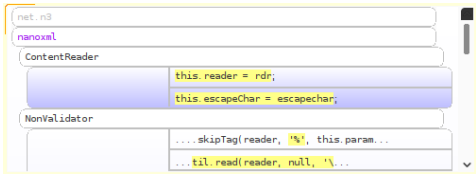
Problem definition. ROSE assumes the developer is using the debugger and the program is suspended with an observed *problem* caused by a semantic *error*. The developer invokes ROSE at the line where the program stopped. ROSE starts by asking the developer to quickly specify the problem *symptoms* (e.g., an unexpected exception). Note that in this step ROSE asks the developer to describe the observed problem caused by the error, but not the error itself. In the absence of test



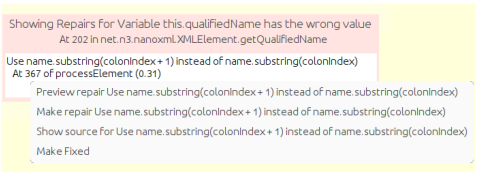
(a) Problem specification options.



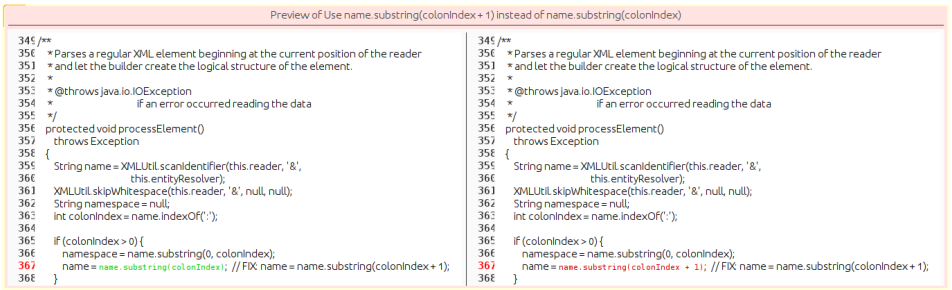
(b) Detailed options for the variable value problem.



(c) Potential repair locations.



(d) Repair suggestions.



(e) Repair preview.

Fig. 2. User interface of ROSE.

suite, the defined problem is used to localize the error and validate patches. This process is detailed in Section 4.1.

Fault localization. After the problem is defined, the developer asks ROSE to suggest *repairs*, code changes that will fix the original problem. ROSE first does fault localization to identify where the error might be. In the absence of a test suite, and without rerunning the program, ROSE uses an abstract-interpretation-based flow analysis to statically compute a partial backward slice that starts from the stopping point, and uses the result of this slice to identify potential lines to repair. The slice is partial because ROSE takes into account the problem symptoms and the current execution to create an accurate slice for the identified symptom, while limiting the slice based on execution distance from the stopping point. More details about fault localization can be found in Section 4.2.

Baseline execution generation. Before invoking the patch generators to generate candidate repairs, ROSE creates a baseline execution that duplicates the original problem. The baseline execution not only serves as a foundation for validating repairs that ROSE will generate in the following step but also helps filter out the potentially faulty lines that were not actually executed.

In general, a full problem-duplicating execution might have involved user or external events and can be too complex to obtain. To ensure efficient and practical repair validation, ROSE considers only the execution of an error-related routine on the current call stack and everything it calls. ROSE identifies a suitable error-related routine for which a partial problem-duplicating execution would be relatively easy to create. ROSE guarantees that there is no routine higher up the call stack that includes other potentially faulty locations to ensure that all possible locations that can be considered are considered. This might not include all the locations identified by fault localization but will include all those whose changes ROSE can detect. ROSE also resets the execution environment internally for the start of this execution. Next ROSE obtains the complete execution history of that routine, and finds the location corresponding to the current stopping point in the execution history. Because fault localization is based on a static analysis, there can be identified potentially faulty locations that are not executed in the baseline execution and would therefore not cause detectable changes. These are eliminated from further consideration. This whole process is described in Section 4.3.

Repair generation. ROSE next invokes the patch generators that have been plugged into the framework to generate potential repairs for each identified location. ROSE includes a set of default pattern-based suggesters and has been extended with suggesters based on code search and machine learning. Each suggester is responsible for generating the repair edit, either as a text edit or an edit of the underlying abstract syntax tree, providing a brief description of the repair for the developer, and providing an estimated priority for the repair at this location. Section 4.4 details how ROSE uses the suggesters to generates repairs.

Repair validation. ROSE next validates the suggested repairs using the baseline execution. This is done by comparing the full trace of the baseline execution with the corresponding trace of each repaired program. ROSE takes into account the problem symptom and a variety of matching situations to compute a semantic priority score approximating the likelihood of the problem being fixed and the repair being valid (i.e., correct and non-overfitting [104]). The semantic priority score is used in conjunction with the syntactic priority derived from repair generation to create a final priority score for the repair. This is described in Section 4.5.

Repair presentation. Finally, ROSE presents the potential repairs to the developer as in Quick Fix [96] and previous interactive test-suite based APR tools [37, 46]. It shows the repairs in priority order and limits the presentation to repairs that are likely to be correct. The repairs are displayed as they are validated so that the developer can preview or make a repair as soon as it is found. Section 4.6 details this.

3 Related Work

In this section, we discuss related work in test-suite-based and non-test-suite-based program repair, fault localization, patch validation, and debugging.

3.1 Test-Suite-Based Program Repair

Most of the current automated program repair (APR) techniques are test-based. They use a test suite as the correctness criterion to help localize where the bug might be, guide the patch generation, and validate the plausibility of the patches (i.e., check whether the patched programs pass all the tests). These techniques differ by the strategies used to produce repairs, and are generally classified as search-based, pattern-based, semantics-based, and learning-based approaches.

Program repair can be viewed as a search problem. A search-based approach implicitly defines a space of patches and aims to explore the space via some strategy to find a patch that correctly fixes the bug. A group of the systems including GenProg [59], AE [116], ARJA [140, 141], and HDRepair [58] use evolutionary algorithms such as genetic programming to perform the search.

These systems define mutation and crossover operators to transform the original program, create mutants as patched programs, and use fitness functions to guide the search of finding likely correct patches. Some techniques leverage the code similarity to identify potential fix code for patch generation. ssFix [127] and sharpFix [128] search for and reuse fix code that is syntactically similar to the faulty code within the original program and across other projects in a code database to produce patches. SOSRepair [2] and SearchRepair [49] compare program semantics distilled in the form of constraints via symbolic execution to find behavior-similar programs for bug-fixing. SimFix [39] focuses on identifying similar code from the local project to produce patches. It additionally leverages the frequent modification knowledge mined from a corpus of bug-fixing patches to reduce the search space. CapGen [118] performs three types of context-aware code analysis to prioritize mutation operators and patches. Hercules [101] analyzes code context to identify similar-looking evolutionary siblings for simultaneous repair. TransplantFix [133] leverages the inheritance relationship to identify fix code from the local program and perform graph-differencing-based transplantation to apply the fix code to the faulty place for repair.

Many APR techniques use fixing patterns for patch generation. PAR [50] is an early system that uses 10 pre-defined patterns distilled from over 60,000 bug-fixing instances to produce patches. ELIXIR uses eight patterns including adding a null-pointer checker and changing the boolean operator for an infix expression. Liu et al. [67] analyzed the patterns used by a set of previous techniques, identified the frequent ones, and created TBar [67], a state-of-the-art technique that systematically applies 15 categories of patterns to generate patches. In addition to APR systems that use a small number of pre-defined patterns, others [54, 69] aim for inferring and mining patterns for effective patch generation. Recent efforts have investigated using deep learning to guide pattern selection and patch generation [81, 82].

Some of the semantics-based approaches achieve automated repair by performing semantic analysis such as symbolic execution to obtain constraints encoding the expected behavior for the faulty parts of the program and then doing code synthesis to generate the correct code [57, 79, 80, 88, 119]. Apart from the constraint-based techniques, others use a value-based specification derived from its semantic analysis. They search for the expected values, or angelic values, that the faulty expressions should hold as the correct behavior, and then work on performing code synthesis to generate the fixed code satisfying the specification [70, 71, 130, 132].

Many of the learning-based approaches treat code repair as a neural-machine-translation (NMT) problem [41, 42, 74, 137–139, 151]. They leverage the power of deep learning to train an encode-decoder model based on a large corpus of bug-fixing instances. The encoder encodes the input code into hidden status capturing the contextual semantics while the decoder translates the semantics into fixed code with errors corrected. Current NMT-based techniques differ primarily in the code pre-processing methods (which decide the context of faulty code considered for translation, the abstraction of the context, and tokenization), the code representation (based on for example sequence, tree, and graph), and the model (e.g., LSTM, RNN, and GNN). Readers can refer to [144] for more details.

Most recently, researchers looked at using Large language Models (LLMs) for repair. Xia et al. proposed AlphaRepair [123], which uses the pre-trained CodeBERT model to perform cloze-style APR that predicts the fixed code for the faulty code masked out. Later, they also developed FitRepair [121], which follows the plastic surgery hypothesis and combines domain-specific fine-tuning and relevant-identified-based prompting to produce patches, and ChatRepair [124], which interacts with ChatGPT [15] in multiple iterations and incorporates test failure information to improve repair performance. Wei et al. proposed Repilot [115], an approach that fuses LLM with the completion engine to produce patches. Nashid et al. [87] and Wang et al. [113] proposed approaches that search for relevant bug-fixes and use them as examples to guide LLM-based patch generation.

Peng et al. [94] investigated using templates for LLM-based repair of Python type errors. Silva et al. [103] proposed RepairLLaMA, which combines an effective code representation and the efficient LoRA adaptor to fine-tune the LLaMA LLM for program repair.

Several studies have also been conducted to understand whether and how LLMs can be reused for APR. Prenner et al. [95] showed that OpenAI's Codex can repair many errors in the QuixBugs dataset and is competitive with NMT-based techniques. Similarly, Sobania et al. [105] found that ChatGPT is also effective to deal with QuixBugs errors. Fan et al. [22] tried to understand the failure of Codex for bug repair and explored the possibility of combining APR with Codex for better repair. Xia et al. [122] used 9 pre-trained LLMs to repair hundreds of real and programming-level errors from five existing benchmarks in different settings addressing single-function, single-hunk, and single-line repairs and showed that the LLM-based tools outperform other learning-based and traditional techniques. Jiang et al. [40] also showed that LLMs can fix more bugs than other deep-learning-based tools and that fine-tuning these LLMs can lead to more effective repair. Cao et al. [13] investigated how effective ChatGPT was at repairing deep learning program errors. Zhang et al. [145] studied the effectiveness of ChatGPT in repairing a new bug dataset containing 151 errors for 15 programming problems. Finally, Huang et al. [32] investigated using fine-tuned LLMs with different combinations of code abstractions, code representations, and checkpoint methods used to address software bugs, programming errors, and vulnerabilities. They identified the fine-tuned LLM with the best combination and showed that LLM is also promising to handle complex multi-hunk bugs. While various LLMs have been investigated to fix errors, they are actually only used as patch generators. For repair, current LLM-based techniques assume that the faulty code (a method, hunk, or line) has been effectively identified and rely on a test suite to validate the patches generated.

Unlike the above approaches, ROSE is a repair framework rather than a technique. It allows an APR patch generator to be easily integrated into an IDE to provide quick and effective debugging support. A ROSE-based tool, which employs a specific patch generator to suggest repairs, is fundamentally different from existing test-based techniques in that it is interactive, works in an IDE, and does not assume the existence of a test suite for fault localization and patch validation.

3.2 Non-Test-Suite-Based Program Repair

While most APR techniques require a test suite, some do not. A group of test-free techniques focus on addressing security vulnerabilities via methods such as constraint-based code synthesis [25], neural machine translation [17, 23], invariant inference [148], and safety property guidance [33]. Another group targets GitHub issues. Jimenez et al. [43] used four LLMs to generate patches based on the issue description and the project code. SWE-agent [134] and AutoCodeRover [149] use LLMs as agents to analyze the issue, search for code, and propose patches. CrossFix [107] searches for similar GitHub issues or bugs and leverages their fixes for patch generation. A ROSE-based repair technique is different from these techniques in that it is not limited to security vulnerabilities or GitHub issues but instead addresses more general semantic errors that arise in a debugging scenario. Also, unlike these vulnerability-oriented techniques, a ROSE-based technique follows a generate-and-validate procedure and involves test-free fault localization and patch validation in its repair process.

Getafix [6], Phoenix [7], FootPatch [110], and SymlogRepair [68] are repair techniques that rely on static analyzers (e.g., Infer [34]) for bug detection and repair validation. They differ in targeting different types of bugs and using different approaches to generate repairs. Specifically, Getafix performs hierarchical clustering of the past fixes to mine patterns and further selects suitable patterns to produce patches addressing potential issues warned by static analyzers. Phonenix mines patches that fix static analysis violations from a collection of code repositories. It then uses a DSL-based synthesis algorithm to learn repair strategies based on the patches mined and applies

these strategies to make new patches. FootPatch relies on Infer's static analysis to detect three types of pointer-related bugs that are resource leaks, memory leaks, and null dereferences and searches for fix ingredients from the local program to construct patches. To extend the scope of static-analysis-based repair, SymlogRepair uses Datalog as the domain-specific language to define program analysis properties and performs symbolic execution to achieve semantic repair fixing property violations. Unlike these techniques, ROSE does not rely on static analyzers for specification. It does not aim to address potential issues flagged by static analyzers but instead works in an IDE to handle semantic failures actually observed for debugging. ROSE interacts with the developer to obtain a problem description based on which it identifies repair locations and validates repairs.

ROSE is also related to repair techniques using specifications in the form of contracts such as pre and post conditions [93, 114], bug reports [55, 66, 86], the Alloy specifications [28], and reference implementations [78]. ROSE differs from these techniques in that it does not require these types of more formal specifications.

Finally, ROSE is related to techniques [9, 30, 63] that require human interaction for program repair. Shipwright [30] involves developer interaction to repair broken Dockerfiles. InPafer [63] asks for a developer's feedback to filter incorrect patches. Learn2Fix [9] trains an automatic oracle to guide the repair process via multiple queries provided by the developer. Unlike these techniques, ROSE's goal is not to repair specific types of bugs, filter incorrect patches, or train an oracle for repair. It aims to provide quick repair suggestions for semantic errors encountered while debugging. ROSE interacts with the developer only once at the beginning to obtain a problem description and then automatically generates repair suggestions without further interaction.

3.3 APR Application in Industry

There has been successful APR deployment in industry. SapFix [76], which is built on top of Sapienz [75] for crash detection, handles the fix of Android app crashes and has been deployed into Meta's continuous integration system. When Sapienz reports a crash, SapFix uses both template-based and mutation-based strategies to propose patches and later tests the patches with the help of Sapienz. Patches that pass the tests are given to an engineer who is qualified to evaluate the patches for review. GetaFix [6] leverages previous fix patterns to suggest fixes addressing static analysis warnings. It was deployed to Meta for suggesting fixes for issues detected by the Infer analyzer. Empirical evidence shows that developers accepted 42% fixes suggested by GetaFix. BloomBerg also attempted to incorporate APR into the software development process. It uses the Fixie tool [51] aiming to tackle simple, repeated bugs and save the debugging time for developers. Fixie addresses a few common bug types and for each extracts fix patterns from previous commits for bug fixing. Pracfix [147] is another pattern-based tool that extracts generic reusable patterns from historical code changes for patch generation. It has been applied to software development in Alibaba and is shown to be liked by the developers. ROSE is related to these techniques in that it aims to help developers debug and supports fast repair suggestion. Unlike these tools, however, ROSE requires an interaction with the developer to get a specification of the problem, and it is not designed to tackle static errors (flagged by static analyzers for example) or any specific type of semantic errors (such as crashes). It is a framework that supports non-test-based patch validation and allows the integration of different patch generators.

3.4 Fault localization

There has been a large body of research [120] dedicated to finding program elements that trigger a failure. Many of the existing fault localization techniques are spectrum-based [21]. They assume the existence of a test suite and analyze the code coverage derived from the execution against the

passing and failing test cases in the test suite to infer the suspiciousness of the program elements. Various approaches have been proposed [120] to compute code suspiciousness based on different forumlae. Studies such as [1, 126] have been conducted to investigate their effectiveness from both the theoretical and practical perspectives. In addition to spectrum-based techniques, there are also other approaches that perform mutation-based analysis [31, 85, 91], delta-debugging [19, 142], value replacement [38], predicate switching [146], information retrieval [56, 100, 112, 150], version history analysis [117], program repair [72], model-driven analysis (e.g., [102]), and various learning-based strategies [61, 62, 73, 81] to achieve fault localization. ROSE's fault localization differs from these approaches in that it is flow-analysis-based and does not require test suite or program re-execution to identify suspicious locations to repair. Its fault localization is also different from standard slicing-based approaches [3, 4] and their variants (e.g., [25]) in that it works in a development environment; it uses a fast, continuous flow-analysis to efficiently compute a static backward slice that simulates the real dynamic slice; and that it takes into account the abstract program states, the problem symptom, and current execution environment to limit the slice to provide accurate results.

3.5 Patch validation

Most APR techniques rely on a test suite and require rerunning the program for validation [29]. Given a set of patches, their validation approaches first apply the patches to the original program to obtain the patched programs and then compile and run these programs against the test suite to decide whether any of the patches can pass all tests and are possibly correct. Patch validation is a process that involves repeatedly executing the program against the test suite and is generally considered an expensive step for bug repair. To accelerate validation, advanced techniques have been proposed. UniAPR [16] performs on-the-fly execution to use only one JVM process to validate patches and avoids repeatedly invoking new JVM processes to save time. VarFix [119] uses variational execution that allows merging program edits and creating meta-programs to accelerate the search and validation of patches. ExpressAPR [125] uses five strategies based on mutant schemata, mutant deduplication, test virtualization, test prioritization, and parallelization to achieve fast patch validation. Other repair approaches resort to non-test-based specifications including the reference implementations [78], the contracts [114], bug report description [55], static analyzers [6, 7], specific program behaviors (e.g., crash [24]), and program properties (e.g., heap properties [110]) to achieve validation.

Unlike these approaches, ROSE is not test-based. It does not require program re-execution against the tests for validation, nor does it require the various forms of specifications as discussed above. ROSE's validation is done by producing and comparing simulated traces that reflect the dynamic failure-exposing executions of the original and repaired programs to decide whether a patch resolves the problem specified by the user and infer its correctness.

PATCH-SIM [129] is related to ROSE in that it also compares traces to infer patch correctness. The approach is however different from ROSE's in threefolds. First, PATCH-SIM requires a test suite and its correctness inference is based on the comparison of traces derived from executions of original and patched programs over both failing and passing test cases whereas ROSE is not test-based and its trace comparison is based solely on the failure-related executions. Second, PATCH-SIM compares complete-path spectra derived from real dynamic execution whereas ROSE focuses on comparing failure executions that are simulated and reflect the real executions. Third, PATCH-SIM compares traces only to compute a distance score quantifying the trace similarity. ROSE is different in that it compares executions stepwise, considers both program control flow and program state, and takes into account the problem symptom and a variety of execution matching situations to infer patch correctness.

3.6 Debugging

Debugging has long been recognized as a laborious and time-consuming task. To make it easier, various automated debugging approaches have been proposed. Fault localization [120] is generally regarded as an automated debugging approach, since it can potentially save developer time by automatically pinpointing the buggy code for developers to examine and fix. Other approaches provide debugging aids by for example allowing developers to ask why and why-not questions and providing guidance to answer them [52], incorporating user feedback that confirms the (in)correctness of variables and paths to narrow down the execution steps that are suspicious [65], re-computing code suspiciousness based on a factor graph with the user feedback provided [131], and facilitating explainable scientific debugging with the help of LLMs [47]. Compared to these approaches, ROSE is different in that it serves as an interactive repair framework that can be integrated into an IDE to provide actual repairs.

Modern IDEs such as Eclipse and Visual Studio provide the auto-correction facility [5, 96] to help developers repair simple syntax errors detected by a compiler. A key limitation of such auto-correction tools is that they do not handle semantic errors exposed at run time. ROSE is designed to address semantic errors and is thus different. Similar with ROSE, AutoFix [92] is also shown to be able to provide automated repair support within an IDE. Unlike ROSE, however, AutoFix is not interactive and requires either a contract (pre- and post-conditions) or a test suite as the specification. It is restricted to dealing with Eiffel programs and is not a repair framework.

4 Approach

In this section, we elaborate on the six steps that ROSE takes to suggest repairs for debugging.

4.1 Problem Definition

ROSE assumes the program is suspended in the debugger at a location where unexpected behavior is observed. The developer invokes ROSE from this stopping point. ROSE tries to infer the problem first and then allows the developer to approve or change its inference. If the program is stopped due to an exception, ROSE assumes that the problem is that the exception should not have been thrown. If the program is stopped at an assert statement or call that failed, ROSE assumes that the assertion failure is the problem. In other cases, ROSE assumes that execution should not have reached the current line. This might occur if the line represents defensive code that checks for an unusual or erroneous situation at which the developer had set a break point. The developer can also indicate that a variable or expression at the stopping point has the wrong value (Figure 2a). In this case, ROSE shows the current value and lets the developer specify either a correct value or a constraint on the correct value such as non-null or greater than 0 in the "Other Value" box (Figure 2b). The default is simply not equal to the current value.

For its later phases, in particular fault localization, ROSE translates the various conditions into more specific internal checks on variables or expressions. If the symptom is a thrown exception, ROSE will determine the expression causing the exception and include it as part of the internal problem symptom. For assertion failures, ROSE will determine the expression computed by the code that is part of the assertion and include it. For a variable or expression with the wrong value, it just uses that variable or expression.

Once the developer has defined the problem symptoms, they can ask ROSE to suggest appropriate repairs. Again, for suggesting repairs, ROSE only needs a description of the problem symptoms. The developer does not need to know the error or its location.

Algorithm 1 Fault localization algorithm sketch.

```
1: procedure FindFaultyLocations(Location loc,Reference r,ValueSet v)
2:     if r == null then ctx = [ r->10, COND->4 ]          ▷ COND is a special reference to control flow
3:     else ctx = [ COND->4 ]
4:     Add ValueSet v to ctx
5:     queue.add(<loc,ctx>)                                ▷ queue is a global variable
6:     ProcessQueue(queue)
7: procedure ProcessQueue(queue)
8:     while queue is not empty do
9:         <loc,ctx> = queue.pop()
10:        ComputeNext(loc,ctx)
11: procedure ComputeNext(Location loc,Context ctx)
12:    if loc has been processed before then
13:        ctx = merge ctx with prior context
14:        if ctx == prior context then return
15:    if loc is start of a method then
16:        callctx = new call context with any locals in ctx mapped to arguments on stack at call site
17:        Decrement callctx COND priority                 ▷ Decrement the priority associated with COND
18:        for each call site of this method consistent with stack do
19:            Let callloc = location of call
20:            queue.add(<calloc,callctx>)
21:    else
22:        for each prior location priocloc of loc do
23:            if priorloc is consistent with current values in context then
24:                HandleFlowFrom(priorloc,ctx,loc)
25: procedure HandleFlowFrom(Location priorloc, Context curctx, Location curloc)
26:    BackflowData bfd = computePriorStateContext(curctx,curloc,priorloc)
27:    Context priorctx = bfd.prior_context
28:    if loc is a method call then add relevant arguments to priorctx
29:    for each AuxReference aref in bfd.aux_references do
30:        refctx = create context with [aref->Max of priority in curctx - 1]
31:        if refctx is relevant then queue.add(<aref.location,refctx>)
32:    if priorloc is a method call and method is relevant to the call stack then
33:        for each method m called from priorloc do
34:            returnctx = new context containing return value and COND
35:            queue.add(<return location of m,returnctx>)
36:    if priorctx is relevant then
37:        queue.add(<priorloc,priorctx>)
38: procedure ComputePriorStateContext(Context ctx, Location curloc, Location priorloc)
39:    Let newctx = [ ], arefs = { }
40:    for each <ref->priority> in ctx do
41:        BackFlow bf = ComputeBackFlow(priorloc,curloc,ref)
42:        if bf.reference != null then add [bf.reference->priority] to newctx
43:        Remove computed values from value set in context
44:        for each AuxReference aref in bf.aux_references do
45:            if aref.location == priorloc then add [aref.reference->priority-1] to newctx
46:            else add aref to arefs
47:    if ctx contains COND->priority then
48:        BackFlow bf = ComputeBackFlow(priorloc,curloc,null)
49:        for each AuxReference aref in bf.aux_references do
50:            if aref.location == priorloc then Add [aref.reference->priority-1] to newctx
51:            else add aref to arefs
52:    return new BackFlowData with newctx and arefs
53: procedure ComputeBackFlow(Location fromloc,Location toloc,Reference ref)
54:    Let newref = null, arefs = { }
55:    Consider the code executed between fromloc and toloc
56:    if ref is computed by the code then
57:        if ref is loaded from local/field/array then newref = reference to source of load
58:        else add any parameters of the computation as arefs
59:    else if ref is a stack reference then newref = updated stack reference based on stack delta of instruction
60:    else newref = ref
61:    if ref is empty and code is a conditional branch then
62:        add any parameters used on condition as arefs
63:    return new Backflow with newref and arefs
```

4.2 Fault Localization

In this step, ROSE aims to identify lines that might be causing the identified problem symptom. It assigns priority to these lines based on the execution distance (essentially the distance in a program dependence graph) between the line and the identified symptom. It does this by computing a partial backward slice specific from the problem at the developer's stopping point using values from the debugger and the call stack.

ROSE uses the abstract interpretation-based flow analysis tool FAIT [97] included in the IDE. FAIT updates control and data flow information incrementally as the programmer types. The result of the abstract interpretation is a data structure that associates an abstract program state (stack, variable, and memory values) with each execution point, links from each execution point to its predecessors, the set of methods invoked from each call site, the set of calling points for each method, the set of writes for each field, and the set of writes to each array.

Overview of the approach. ROSE uses this data structure to do fault localization in two steps. It first uses FAIT to construct a limited dependency graph representing a backwards slice from the stopping point based on the identified symptoms. The nodes of this graph are program points from the abstract interpretation and the edges represent either data dependencies for the symptom or control dependencies that affect the symptom. This graph is restricted using the current execution context and the number of changes to the affected variables. Next it finds the set of source lines that are in the graph. These lines are the potential fault locations.

ROSE builds the dependency graph by starting at the stopping point and working backwards, one instruction at a time, using the links from flow analysis. For each program point, it maintains the set of relevant references. References can be to local variables, stack values, or local fields. The initial set of references is based on the values identified by the symptom (the incorrect variable, the expressions causing an exception, etc.) determined in the problem definition stage. At each step, ROSE maps the current set of references to the corresponding set of references at the prior program point, updating stack locations and taking into account computations, reads, and writes.

For each reference, ROSE maintains a priority representing its computational distance from the starting point and a value if it is known from the debugger environment. ROSE also maintains a call stack from the debugger environment. The priority is decremented each time the corresponding reference value was computed and references are discarded when the priority goes to zero. This provides the computational distance limit for the dependency graph. The known values are compared to flow-based constraints to restrict which paths to consider. The call stack is used to restrict which calling sites to consider. A special reference (COND) and priority are used to indicate that control flow should be considered even if it does not affect any current references.

ROSE handles various special cases. For field accesses relevant to a reference, it considers all corresponding field writes as previous program points. For array access, it considers all corresponding array writes. At a method invocation it determines if the method is relevant to the current set of references and, if so, adds all return points as previous program points. At the start of a method, it uses all calling points consistent with the current call stack as previous program points.

The algorithm. An abstract of localization algorithm is shown in Algorithm 1. The initial set of program references is determined by the problem symptom. For example, a problem based on a variable includes a reference to the variable, and a problem based on an exception includes a reference to the stack element containing the expression value causing the exception. The initial context also indicates that control flow is relevant. The set of initial values is obtained from the debugger. The algorithm uses a work queue mechanism to process each location and associated context (lines 8-10).

For a given location and context, the algorithm needs to compute any prior locations that would be executed immediately before the current one, and determine the proper context for that location based on the current context. This computation is based on what code was executed between the prior and the current location. If the current location corresponds to the start of a method, then the algorithm considers all callers to that method in the static analysis that match the current call stack and sets up a new context at each call site, mapping any local parameters to the arguments at the call, and decrementing the condition count (lines 15-20). Otherwise, it uses the flow analysis to determine prior program points where the abstract values are consistent with any current known values and considers each in turn (lines 21-24). Consistency is based on flow constraints which can indicate if a value can or must be null or provide ranges for integer values. These are checked against the known values maintained in the context.

The algorithm considers what was executed between the prior state context and the current one in order to determine the proper set of references for the prior state. ROSE includes code to determine, for a given reference, both what that reference was at the prior state and what other references might have been used to compute it (lines 54-63). A stack reference might change its location in the stack, might disappear, or might become a reference to a local variable (on a load); a local reference might be changed to a stack reference (on a store). A field load adds all locations where that field was stored as auxiliary references; an array load adds all locations where array elements were set. A stack reference computed using an operator, adds the operands as auxiliary references. For example, if the top of the stack was relevant and the code was an ADD, then the reference to the top of the stack would be removed, but auxiliary references to the top two stack elements (the operands of the ADD) would be added with a decremented priority. This information is combined for each reference in the context (lines 40-46) and for control flow if relevant (lines 47-51). The resultant contexts and locations are then queued for future consideration (lines 29-31 and 36-37).

Method calls are handled explicitly by the algorithm. A method is relevant if its return value or the 'this' parameter is relevant. If the call is relevant, the algorithm adds all the parameters to the prior state (line 28). If control flow is relevant, it also queues the return site of any method called at this point with a context that includes the return value if that was relevant (lines 32-35).

The result is a set of program points where a relevant value or relevant control flow was computed. This is the set of potential error locations to be considered. The priority of the location is determined by the maximum priority associated with a reference in the context. This priority reflects the execution distance of the corresponding location, which does not necessarily correlate with fault localization and is currently not used. From the set of program points, ROSE computes the set of potential error lines. It can optionally exclude from this set any test routines or drivers to ensure that ROSE fixes the problem rather than changing the test.

We note that this type of analysis yields different results than control-flow or spectra-based methods. Spectra-based localization, because it uses coverage information, only provides information at the basic-block level. ROSE identifies specific expressions that can directly affect the problem. Statements that are not indicated as erroneous by ROSE do not affect the problem. On the other hand, locations where a new statement should be inserted in a particular branch can be identified by spectra-based techniques but may be ignored by ROSE. ROSE assumes that the latter case can be deferred to repair generation. ROSE's fault localization, because it is based on static information, can include lines that are not executed in the faulty run and are therefore not relevant. These would not be included in spectra-based localization. ROSE deals with this by using the baseline execution to remove these lines.

4.3 Generating a Baseline Execution

After fault localization, ROSE generates a baseline execution that duplicates the problem. The baseline execution is a simulated execution that shows how the program got into the current state based on the current debugging environment. ROSE uses this execution to identify and exclude previously-localized locations that are not executed in the faulty run. These locations will not be considered for repair generation in the following step, as they are not related to the observed failure. The more important use of the baseline execution is for repair validation. Without assuming the availability of test suite and the ability to rerun programs, ROSE validates a repair by obtaining the simulated baseline execution and the corresponding repaired execution and comparing the traces derived from executions to check if the problem symptoms go away, the repair is likely to be correct [104], and the program seems to work.

With the help of SEEDE [98], a practical live-programming facility, ROSE produces a simulated execution trace that includes the history of each variable, field, and array element that have been updated during the execution in terms of the original and new values and the time stamp. The trace also includes a time stamp for each executed line and function call. ROSE could invoke SEEDE to obtain a complete execution e starting at the beginning of the program to recreate the current execution environment. This is not practical because ROSE could be invoked in the middle of a run that involved user or other external interaction, from a run that had gone on for several minutes or hours, or from a thread other than the starting thread. To address this, ROSE tries to obtain a local partial re-execution $\hat{e}$ that starts with a routine on the current execution stack using the current environment in order to duplicate the problem. Using $\hat{e}$, which is shorter than e, for validating potential repairs is more practical and efficient. To find $\hat{e}$, ROSE needs to (1) determine the routine at which to begin and (2) ensure that the simulated execution from SEEDE matches the actual execution.

To determine where to begin execution, ROSE first identifies a close stack frame that includes all potential locations identified by fault localization directly or indirectly, preferring logical starting points, for example, a callback from a system routine. Choosing such a frame handles the case where the stopping point is in one routine, but the error is in one of the callers of that routine or in a routine one of the callers invoked. The user interface ROSE provides also lets the developer specify a particular starting frame.

Ensuring that the simulated execution matches the actual execution can be challenging when the code changed a parameter or global variable the execution depended on. In general, it is not possible to reverse the execution back to the chosen starting point. However, this is often feasible in practice. If the program changes a parameter value it is often possible to recompute the parameter in the calling context. Another typical situation occurs when the program checks whether a value has previously been computed by trying to add it to a *done* set, and just returns if it has. If the actual execution added a value to that set, then the simulated execution would just return and would not match the actual one. SEEDE provides a mechanism for defining additional initializations before creating the execution which ROSE uses to tackle these problems. ROSE attempts to iteratively find appropriate initializations of changed variables. If this approach fails, ROSE will not be able to validate any repair and it yields no results.

Choosing a higher-level starting point involves a trade-off between effectiveness and efficiency. Higher-level starting points will often initialize the problematic values directly, thus eliminating the need to reverse the execution. This is why ROSE prefers starting at the main program or callback from a system routine. However, a higher-level starting point can greatly increase the validation time, making ROSE impractical for interactive use.

Once a baseline execution has been found, ROSE uses the set of lines that were executed in that baseline to restrict the set of possibly faulty locations. It removes any faulty location that is not executed from the set of locations to check. This effectively converts the static analysis based on flow information into a more dynamic analysis.

4.4 Repair Generation

The next step is to generate potential repairs for each identified potential error location. Like Quick Fix or PRF [26], ROSE supports pluggable repair suggesters. Each suggester is invoked for each identified line to generate a set of repairs. For each generated repair, the suggester returns either a text or an abstract syntax tree edit, a description of the repair for the developer, and a local priority for the repair that estimates its likelihood at the particular location. A common routine is provided to generate a default description based on an analysis of the edit. Suggesters are given a priority which is used to determine the order to consider them. The priority is used to run the various suggesters on the different faulty lines in parallel with multiple threads using a thread pool and a priority queue. This ensures that the suggesters most likely to succeed are run early in the process so that successful repairs are reported quickly.

ROSE provides an initial set of suggesters based on patterns. Additional suggesters based on existing APR techniques have been added including a code-search-based approach, a learning-based approach, and an approach based on ChatGPT. We chose these suggesters as we found that they have complementary repair abilities and are fast enough to be used interactively. This lets ROSE offer quick and effective repairs while debugging in the IDE.

Pattern-based suggesters. The suggester with the highest priority looks for code smells [109] (common programming errors) that may occur at a potential error line. This is a pattern-based analysis where the set of patterns is based on various studies of common errors [11, 14] and bug fixes [12, 90] as well as our own programming experience. Current patterns include ones dealing with equality such as using == for =, using == rather than *equals*, and using == for an assignment; ones dealing with strings such as using *toString* rather than *String.valueOf* and calling methods like *String.trim* but not using the result; ones dealing with operators often used incorrectly such as xor or complement; and ones dealing with confusing integer and real arithmetic.

The suggester with the second highest priority is one that handles avoiding exceptions by inserting or modifying conditionals around the code where an exception was thrown, a common fix. Most of the remaining pattern-based suggesters currently have the same lower priority. These include one that handles conditionals by changing the relational operator; one that considers different parameter orders in calls; one that looks for common errors with loop indices in *for* statements; and one that considers replacing variables or methods with other accessible variables or methods with the same data type.

Search-based suggester. ROSE uses sharpFix [128], a code-search-based tool, to find suggested repairs based on similar code in the current project. sharpFix identifies similar code by matching tokens extracted from variable, type, and method names, renames the variables, types, and methods used in the similar code, identifies related statements and expressions, and performs four types of modifications to produce patches. sharpFix has slightly lower priority since it is slower than the simple, pattern-based suggesters, but has proven effective. The results from sharpFix are reordered and filtered to avoid duplicating the other suggesters.

Learning-based suggester. Another suggester uses SequenceR [18] to produce repairs. SequenceR is a machine-learning-based tool that performs sequence-to-sequence learning to "translate" an error line into a fixed line. ROSE invokes SequenceR to produce repairs and again filters and sorts the results to avoid duplication. The current implementation of SequenceR is relatively slow and is prone to generating repairs that do not compile, so this suggester has the lowest priority.

```
You are an Automated Program Repair Tool for Java programs.

The following code contains a buggy line that has been removed.
```
public static int gcd(int a, int b) {
        if (b == 0) {
                return a;
        } else {
>>> [INFILL] <<<
        }
}
```

This was the original buggy line which was removed by the infill location.
`                return gcd(b, a%b);`

Please provide the correct line at the infill location, and nothing else.
```

Fig. 3. An example of the prompt produced by ROSE's ChatGPT-based repair suggester.

ChatGPT-based suggester. A final suggester uses a query to ChatGPT that includes the method containing the potentially faulty line and asks ChatGPT to suggest a replacement for that line. This is invoked once for each potentially faulty line. The suggester produces a prompt to ChatGPT consisting of four parts: (1) a statement telling ChatGPT to perform as an automated repair tool; (2) the method containing the potential repair location (the error line) where the location is replaced with a special mark ">>> [INFILL] <<<"; (3) the original error line (to be fixed); and (4) a request of replacing the mark with error-repaired code. The suggester sends the prompt to ChatGPT to obtain repaired code. Figure 3 shows an example of the prompt for repairing the GCD error in the QuixBugs dataset [64].

For the cases we have considered, this suggester seems quite effective, so we assigned it a high priority. We note that ChatGPT, though widely recognized as an effective repair generator, can generate invalid repairs that are partial statements (e.g., an partial if-statement with only the if-condition and without the then branch); that involve no changes; that are syntactically invalid; or that are insertions rather than replacements. The current suggester handles some of these issues, ensuring the suggested fix is a full, syntactically correct statement when possible and checking if the replacement should be considered as an insertion rather than a replacement. Additional improvements are left as future work.

Other suggesters. The current system and priorities seem to work reasonably well, producing reasonable suggested fixes in a reasonable time. Using other repair generation approaches [84] to improve the recommendations is quite straightforward.

4.5 Validating a Repair

ROSE validates a repair by comparing the simulated baseline execution with the simulated repaired execution from the same starting point using the same initializations. To obtain the simulated repaired execution, ROSE uses SEEDE to re-execute the code after the repair has been made. This yields a full execution trace, which ROSE compares with the baseline execution trace. The

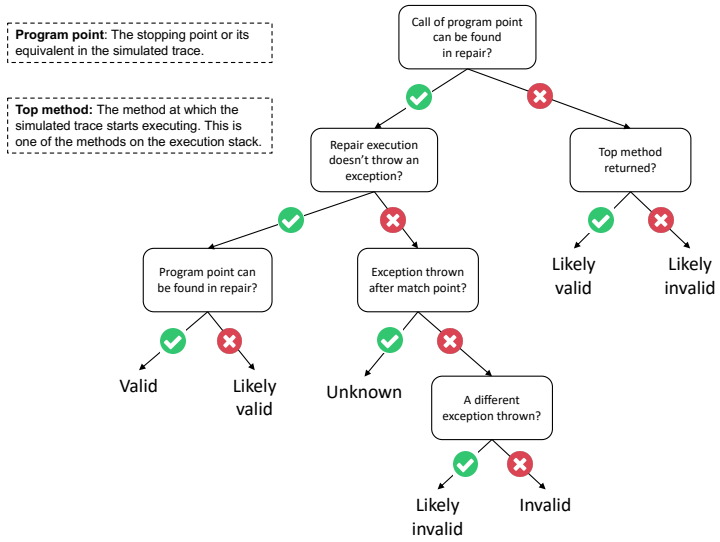


Fig. 4. Repair validation with an exception or assertion problem (i.e., ValidateException/ValidateAssertion in Algorithm 2).

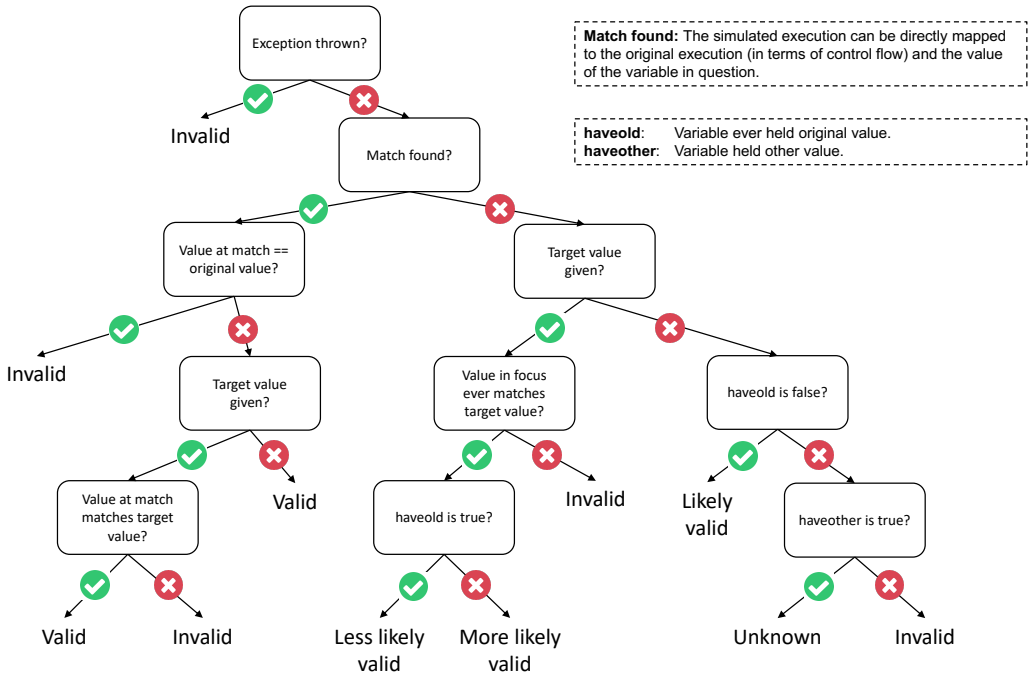


Fig. 5. Repair validation with a variable problem (i.e., ValidateVariable in Algorithm 2).

Algorithm 2 Repair validation algorithm.

```
1: procedure VALIDATE(Execution orig, Execution repair)
2:     if repair has a compiler or run time type error then
3:         return 0
4:     Matcher matcher = COMPUTEMATCH(orig, repair)
5:     if !matcher.executionChanged() then
6:         return 0
7:     double score = VALIDATE<PROBLEMTYPE>(matcher) ▷ ProblemType: Exception, Assertion,
   or Location
8:     double closeness = difference_time / problem_time
9:     return score * 0.95 + closeness * 0.05
```

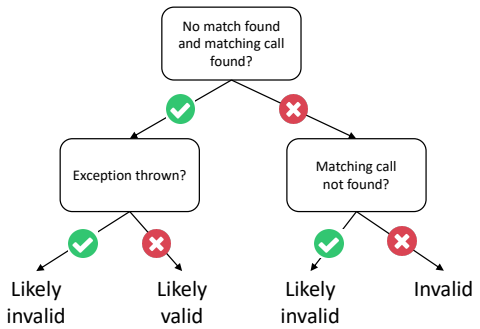


Fig. 6. Repair validation with a location problem (i.e., ValidateLocation in Algorithm 2).

comparison yields a validation score between 0 and 1, with 0 indicating that the repair is invalid and 1 indicating a high degree of confidence in the repair.

ROSE considers the suggested repairs, looking at them in order based on their associated syntactic priority. Multiple repairs are considered in parallel using multiple threads. ROSE keeps track of the number of repairs it has validated and the total execution cost of validating those repairs. It stops considering repairs when either a maximum number of repairs have been considered or when the total cost (number of steps of SEEDE execution) exceeds a limit. These bounds are dependent on whether there has been a likely repair and are designed so that ROSE finishes within a reasonable interactive time frame even if it means not reporting a repair that might otherwise be found.

High-level algorithm. A high-level sketch of the actual matching algorithm is shown in Algorithm 2. If the proposed repair either does not compile or yields a run-time type error, then the repair is considered invalid (lines 2 and 3). Otherwise, the repaired execution is matched with the baseline execution by *ComputeMatch* (line 4). This comparison goes through the two execution traces step-by-step, checking both control flow and data values. It finds the first location where the control flows differ and the first location where data values differ. It also finds the call in the repaired execution corresponding to the call in which the program stopped, and the point in the repaired execution that corresponds to the stopping point in the original execution.

Because repair locations are computed using static analysis, the repaired execution can match the baseline execution up to the problem point. When that happens, ROSE considers the repair invalid as it had no effect (lines 5 and 6). Otherwise, it validates the repaired execution based on the given problem symptoms (line 7) with the various checks shown in Figures 4, 5, and 6.

Validation for exception or assertion problems. If the symptom involves either an unexpected exception or an assertion violation, the validator, as shown in Figure 4, checks if the repaired run avoided the error and scores the repair in various conditions to give high priority to a repair whose execution is close to the original baseline execution but does not throw an exception, particularly, the original exception. Specifically, if the repaired execution includes the problem context, then if no exception was thrown, the repair is considered valid (score is 1) if the problem line was executed or likely valid if it was not. If an exception was thrown after the matching point, then the validity of the repair is not easily known and thus considered equally valid and invalid (score is 0.5); if a different exception was thrown, the repair is considered unlikely; if the same exception was thrown, the repair is considered invalid (score is 0). If there was no matching problem context, the repair is considered probably valid if no exception was thrown and likely invalid if one was.

Validation for variable problems. If the symptom involves a variable value, as Figure 5 shows, ROSE scores the repairs and gives high priority to a repair whose execution is close to the baseline where the variable takes on either (1) the target value consistent with the developer's specified constraint or (2) a value different from the original if no constraint is given. Specifically, ROSE first checks if an exception is thrown in the repaired execution, and if so, treats the repair as invalid. Otherwise, if the program point can be matched in the repaired execution, it compares the repaired value at that point to the target or original value to decide validity. If no match for the program point can be found, then ROSE considers the set of values the variable takes on and computes a score accordingly.

Validation for location problems. In cases where the problem involves reaching an unexpected line, ROSE uses the approach shown in Figure 6 to compute a score based on whether the line was executed. If the program point can be matched in the repaired execution, and the line is still executed, the repair is considered invalid. In other cases ROSE penalizes the executions which throw an exception or do not call the problem method in the first place.

ROSE adjusts all these scores to favor results where the change occurs later in the execution. This is done by considering the point where the execution (either data or control flow) changes relative to the problem point and incorporating this fraction as a small part of the score (Algorithm 2, lines 8 and 9). ROSE later adjusts the score returned by this algorithm by checking if the partial execution completed successfully.

Accurately quantifying the likelihood of repair validity based on the various possible trace matching conditions for effective patch prioritization is challenging. ROSE currently uses a predefined set of constants to quantify likely and unlikely validation scores based on different conditions. These were determined based on an initial suite of problems for testing ROSE and on our debugging experience. We found the patch validation works well in this way (see Section 6.1) and will investigate in future work using more advanced learning-based techniques to possibly obtain better scores for validation.

4.6 Presenting the Result

Once a repair has been validated, it is sent to the front end to be displayed to the developer. ROSE maintains a validity threshold and only sends repairs whose scores are above this. The front end computes a final repair probability by combining the syntactic priority computed by repair generation and the semantic priority computed by validation. It then provides a view of the current repairs ordered by this combined score. This view is updated as repairs are found. This provides a live, on-line view of the potential repairs.

The developer can interact with the repair window as repairs are presented. They can choose a particular repair and ask for a preview. This brings up a window showing the code before and after the repair with the differences highlighted as shown in Figure 2e. Alternatively, the developer can

ask ROSE to make the repair. In this case, using the hot-swap capability of Java, the developer can continue execution with the repaired code.

5 Example

To clarify how ROSE works, consider its behavior on the Defects4J example Chart_1. This bug, found in JFreeChart, involves setting a legend item implicitly and then not having it appear in the collection of legends. We set up a simple main program that creates a renderer, a dataset, and a plot. Then it adds a single value to the dataset in row S1 and column C1. It then expects the legend items from the renderer to be the label S1. However, the set of legend items returned is empty. This is checked with two assert statements in our main program, one to check the size of the set and the other to check if the single element is actually S1.

When we run the example in Code Bubbles, the debugger stops with a breakpoint when the first assertion fails. We then choose "Ask ROSE to help debug" and get the dialog with the assumed symptom being "Assertion should not have failed". This is correct, so we can just click on "Suggest Repairs" to start the ROSE evaluation process.

ROSE first does fault localization. Since the failure is an assertion statement, ROSE notes that an incorrect value is the boolean that results at the end of the assertion expression. It issues a flow query to FAIT asking it to identify all execution points that can affect this value.

The code in FAIT for ROSE, identifies the particular statement in the existing flow analysis, and then does a backward slice from that point noting that the top of the stack (the result of the comparison) is relevant with a score of 10, and control flow to that point is relevant with a score of 4. The backward slice is computed one instruction or execution step at a time, keeping track of which stack and variable values are relevant. The backward slice that results is a graph of 7,010 nodes, each corresponding to a specific execution point, starting with the assertion statement and including lines from our main program, as well as seven other source files from JFreeChart. ROSE condenses these by combining all those that refer to the same line. The result is a set of 608 possible buggy lines.

ROSE next creates a baseline execution for the run. In this case, it executes the test code that sets up the renderer, dataset, and plot and then fails with the assertion exception. The resultant execution trace contains 64 method calls from non-library methods with 443 lines being executed. With variables and values included, the overall trace is over 5000 lines of XML. ROSE first uses this trace to eliminate all potentially buggy lines that were never executed or that were part of the main program. Of the 608 starting locations, 491 were never executed, 69 represented duplicates, and 10 were in the driver program. After eliminating these, 38 possible buggy locations in 6 different files were left to consider.

ROSE next queues up tasks to run each of the 13 possible checkers on each of the 38 potentially buggy locations. The 484 tasks are run in parallel using a thread pool using priorities based on the priority of the checkers. When ChatGPT is used, the various checkers generate 110 possible fixes. However, ROSE detects that 46 of these are actually duplicates, so only 64 possible fixes need to be checked. The correct patch, replacing != with == in a conditional in getLegendItems, is generated both with and without using ChatGPT. ROSE is not perfect at detecting duplicates, so it reports three possible patches when ChatGPT is used, all of which are essentially the same. When ChatGPT is not used, 92 possible repairs are generated but only 46 possible fixes need to be checked, and the correct patch is generated by a pattern-based suggester. Note that the checking order varies between runs because of the inherent parallelism in the checking process.

To validate the results, ROSE again uses SEEDE for each of the possible patches, generating the corresponding trace. Most of these result in an exception, either the original assertion exception, a null pointer exception, or an illegal argument exception. For five to eight of the possible patches,

SEEDE determines that the result will not compile and eliminates them. Two of the possible patches yield run time errors, generally type inconsistencies. Only the correct patch results in a complete run without any exceptions, and is thus considered valid.

The overall process takes about 28 seconds to run, with an initial correct result being returned after about 10 seconds.

6 Evaluation

We implemented a prototype of ROSE that works with the Eclipse-based Code Bubbles IDE [10] and developed two ROSE-based tools using different sets of APR patch generators as the repair suggesters. To evaluate ROSE, we did two studies, an effectiveness study and a utility study. In the effectiveness study, we applied the ROSE-based tools to two benchmarks QuixBugs and Defects4J for error repair. We assessed the repair abilities of these tools with a focus on evaluating the effectiveness of ROSE's core components, fault localization and patch validation. To understand ROSE's usefulness and practicality, we did a utility study by recruiting 26 university students to perform four debugging tasks with and without ROSE. The study was designed to determine if ROSE is helpful to developers.

We wanted to answer four research questions:

- **RQ1**: How effective are ROSE's core approaches: fault localization and patch validation?
- **RQ2**: How efficient are ROSE-based tools for error repair?
- **RQ3**: How many errors can ROSE-based tools repair?
- **RQ4**: How useful is ROSE in helping people debug?

We designed the effectiveness study to answer the first three questions and the utility study to answer the last question.

6.1 Effectiveness Study

6.1.1 Setup. As stated in Section 4.4, the ROSE framework integrates four repair suggesters for patch generation. In the experiment, we created two ROSE-based tools, ROSE-PC and ROSE-PS. ROSE-PC uses a combination of the pattern-based generators and ChatGPT (v3.5-turbo) [15] as the repair suggesters. ChatGPT was shown to have powerful error-repair abilities [105, 124], and pattern-based generators can serve as complementary role [94]. We used ROSE-PC in the experiment to investigate how ROSE works with state-of-the-art patch generators. ROSE-PS on the other hand uses the pattern-based generators and the search-based sharpFix technique [128] for repair generation. We used ROSE-PS to see how ROSE works with traditional (non-LLM-based) repair generators. An evaluation of ROSE with the traditional suggesters helps us assess ROSE's effectiveness without the potential influence of LLM's data leakage issue [144]. We did not include the SequenceR suggester for the study, as we found that the tool is prone to producing a large fraction of invalid patches that do not correctly repair the error.

We chose two widely used benchmarks for APR [77, 136], QuixBugs and a subset of Defects4J (v2.0). QuixBugs [64] is a set of 40 programs containing typical programming errors. Defects4J [45] is a corpus of more complex development errors. For each, we checked how many errors can be repaired and how quickly can this be done. More importantly, we evaluated the effectiveness of ROSE's fault localization and patch validation approaches by analyzing how they contributed to the final repair result, either a success or a failure.

To simulate an IDE-based interactive repair experiment, for the QuixBugs benchmark, we created an Eclipse workspace containing all 40 unrepaired programs but no tests. For each unrepaired program, we created a main program that effectively ran a failing test, identified the problem symptom based on the original JUnit assertion, and ran ROSE to generate repairs. Following a

standard way to evaluate automated repairs, we considered a repair correct if it was semantically the same as that provided in the benchmark (the ground truth repair). While determining semantic equivalence can be challenging, it was clear and easy in these cases.

For the Defects4J benchmark, we focused on overt errors relevant to the assumptions of ROSE. These were errors that involved a single line being modified or added. Our initial set of errors was those with single-hunk fixes used to evaluate SequenceR [18]. We identified different projects, Chart, Closure, Lang, Math, Math3, Mockito, and Time, eliminated multi-line errors, grouped the remaining errors by their base package, created a working set for each package that included the error code if possible, and then eliminated errors that no longer failed in that working set. The result was a set of 56 errors. We further augmented the set of errors with four new errors (Chart14A, Lang47A, Math22A, and Math35A), which were derived from the Defects4J multi-hunk errors in the seven projects and are single-line errors exposed by a failing test. For each of the seven projects, we set up an Eclipse workspace that included code with all the associated errors. Next, we created a main program for each error based on a failing test, replacing the JUnit assertion of the test with an assert statement. We used the failing tests only to identify the problems to repair so the results could be compared to prior work. We ended up with 60 different Defects4J errors. For each, we used the failing assertion or a thrown exception as the problem symptom for ROSE. As with QuixBugs, we ran ROSE on each of the errors in each workspace. We considered a repair suggested by ROSE to be correct if it was the same or semantically equivalent to the Defects4J corrected code at the same location.

Table 1. Result of fault localization (FL) and patch validation (PV). FL: Percentage of errors for which ROSE's fault localization succeeded (i.e., ROSE successfully found the error line); PV: Percentage of repaired errors for which ROSE's patch validation ranked the correct repair as top-1, 3, 5, and 10. Note that there was no error for which correct repair was not found due to the failure of ROSE's patch validation.

Tools	All					QuixBugs					Defects4J				
	FL	PV				FL	PV				FL	PV			
		Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10
ROSE-PC	0.89	0.85	0.96	1.00	1.00	1.00	0.89	0.92	1.00	1.00	0.82	0.81	1.00	1.00	1.00
ROSE-PS	0.89	0.88	0.94	1.00	1.00	1.00	0.86	0.93	1.00	1.00	0.82	0.89	0.95	1.00	1.00

6.1.2 ***Result for RQ1****: Effectiveness of fault localization and patch validation.* ROSE is an IDE-based repair framework intended to make debugging easier. We focus on evaluating ROSE by assessing the effectiveness of its core components, which are the test-free fault localization and patch validation. Table 1 presents our result. Specifically, it shows the percentage of errors (QuixBugs, Defects4J, and both) for which ROSE's fault localization included the line to be patched (the FL columns) and the percentage of errors for which the patch validation ranked the correct repairs ROSE generated within the top-1, 3, 5, and 10 results (the PV columns).

ROSE's fault localization result successfully included the fault location for about 89% of the errors (second column). This shows that ROSE's fault localization found the target repair locations for most of the errors and is effective. For all QuixBugs errors, ROSE identified the right locations to repair (column FL under QuixBugs), and for about 82% Defects4J errors, it found the right locations (column FL under Defects4J). The fault localization failed to find the repair locations for 11 Defects4J errors. For seven of the errors, fault localization failed because the target location is too far away (execution-wise) from the stopping point and is not considered; for two errors, it failed because the error location is related to a field declaration, not an executed statement, a situation that is currently not supported for detection and repair; and finally for the other two errors, ROSE did not find the locations due to misidentifying a dependency from a Java library method that changes the internals of an object, and due to the failure caused by handling an expected exception.

After fault localization, the baseline execution generation phase further eliminated 20-50% of the falsely localized lines, and it never filtered out a correct location. This shows that the baseline generation can help limit the localization result and improve the repair accuracy. Note that ROSE ignores the rank of the error location as reported by fault localization. It generates repairs for all the identified locations and then validates and prioritizes those repairs.

The PV columns of Table 1 show how ROSE performs for patch validation. Unlike fault localization, because the tools ROSE-PC and ROSE-PS generated different repairs, their validation results are different. Our result shows that ROSE's patch validation is highly effective. According to Table 1 (columns 3–6), for over 85% errors, ROSE's patch validation gave a top-1 rank for the correct repair, and the ranks for all correct repairs are within top-5. Note that the average number of candidate patches that ROSE looked at for validation and prioritization is not low and is over 32 for QuixBugs and over 58 for Defects4J errors, according to the *#Checked* columns of Tables 2, 3, and 4. Overall, our result shows that ROSE can effectively prioritize the candidate repairs and hence ROSE can propose a small number of repairs for the user to review, which demonstrates ROSE's practicality for debugging.

Columns *#Results* and *Correct Rank* of Tables 2, 3, and 4 present more details about the validation result in terms of the number of repairs returned by ROSE's backend (*#Results*) and the rank of the correct repair (*Correct Rank*) for all errors the ROSE-based tools correctly repaired.

> ROSE's fault localization and patch validation without test cases are very effective. The fault localization correctly identified the repair location for 89% of the errors. The patch validation gave a top-1 rank for over 85% of errors and a top-5 rank for all. Patch validation was never the reason for ROSE's repair failures.

6.1.3 ***Result for RQ2:*** *Repair Efficiency.* Efficiency is a key factor that affects the practicality and usefulness of ROSE. We measured the running time of ROSE and presented the results in the columns *Total Time* and *Fix Time* of Tables 2, 3, and 4. *Total Time* gives the total time in seconds for all repairs to be found. *Fix Time* shows the time in seconds for the correct repair to be reported and viewed.

Our result shows that the ROSE tools are highly efficient. It takes only seconds for the tools to report the final repair suggestions. For ROSE-PC, the average total time for finding all repairs is only about 4 seconds for QuixBugs errors and 18 seconds for Defects4J errors (row *Average* and column *Total Time* in Tables 2 and 3). The time for finding and reporting a correct repair is much shorter and is only 0.8 seconds for QuixBugs and 5.7 seconds for Defects4J errors (row *Average* and column *Fix Time* in Tables 2 and 3). A delve into the time taken by ROSE-PC (*Total Time*) to repair these errors shows that a significant fraction was used for repair generation and validation (which are bundled together at the implementation level to allow a quick report of valid repairs). On average, about 6.3% of the time was spent on fault localization, 13.7% was dedicated to baseline execution generation, 63.9% was consumed by repair generation and validation, and the remaining 16.1% was for setup, information sending, file loading, etc. According to Table 4, the ROSE-PS tool is also fast and has similar repair time.

For the same QuixBugs and Defects4J errors, running a test suite once takes about 0.5 and 16.5 seconds. Note that because a test-based repair technique often needs to run the test suite multiple times to validate different patches, it is much more expensive than ROSE. Our statistics of various test-based repair techniques shows that these techniques can take minutes to repair a bug (see [20] for more details).

Table 2. Results for QuixBugs errors by ROSE-PC, the ROSE-based tool with the default pattern-based generators and ChatGPT used as repair suggesters. **#Results**: the number of repairs returned by the backend; **Correct Rank**: the rank of correct repair; **Total Time**: the total time for all repairs to be found; **Fix Time**: the time for the correct repair to be reported and viewed; **Fix Count**: the number of repairs validated; **SEEDE Count**: the total number of instructions SEEDE executed before processing the correct repair; **#Checked**: the number of repairs on which validation was run. All times reported are in seconds.

QuixBugs Error Names / Statistics	#Results	Correct Rank	Total Time	Fix Time	Fix Count	SEEDE Count	#Checked
BitCount	6	1	0.6	0.1	1	18	10
BreadthFirstSearch	4	1	5.0	0.9	5	689	20
BucketSort	2	1	3.3	1.1	9	20463	12
DepthFirstSearch	7	1	3.4	0.6	2	194	15
DetectCycle	6	1	2.2	0.2	4	134	26
FindFirstInSorted	11	5	3.3	0.2	6	215	51
FindInSorted	9	1	2.4	0.1	1	56	51
Flatten	5	2	2.2	0.2	1	785	16
Gcd	3	1	1.3	0.1	2	72	7
GetFactors	1	1	2.4	0.8	11	25159	16
Hanoi	4	1	2.9	0.2	8	5334	45
IsValidParenthesization	2	1	1.4	0.1	1	64	9
KHeapSort	1	1	2.1	0.1	1	1642	10
Knapsack	5	1	4.5	0.6	9	42748	73
Kth	15	1	4.6	0.2	2	3039	61
LcsLength	3	1	9.3	0.2	2	17639	57
Levenshtein	1	1	17.5	8.6	11	482935	15
Lis	11	1	3.3	0.3	10	10182	68
LongestCommonSubsequence	2	1	5.3	0.2	4	6877	42
MaxSublistSum	2	1	1.7	0.2	4	464	18
MinimumSpanningTree	8	1	3.9	0.4	2	6793	42
NextPalindrome	1	1	1.9	0.2	11	8602	21
NextPermutation	4	1	3.7	0.5	12	11810	83
Pascal	16	1	4.4	0.4	7	6942	76
PossibleChange	5	1	1.7	0.3	3	2300	12
QuickSort	3	1	4.0	0.3	6	38785	58
ReverseLinkedList	3	1	4.1	2.8	25	63507	28
RpnEval	4	1	2.2	0.2	3	2567	17
ShortestPathLength	1	1	5.4	0.9	7	25978	78
ShortestPathLengths	3	1	13.3	3.7	13	1549196	22
ShortestPaths	2	1	2.8	0.7	9	75649	11
Sieve	3	3	2.0	0.6	6	4770	6
Sqrt	8	5	2.0	0.1	2	36	17
ToBase	1	1	2.0	0.1	1	45	21
TopologicalOrdering	2	1	4.4	1.0	4	11040	31
Wrap	3	1	2.3	0.3	10	23090	23
Median	3.0	1.0	3.1	0.3	4.5	6063.5	21.5
Average	4.6	1.3	3.9	0.8	6.0	68050.5	32.4
StdDev	3.8	1.0	3.3	1.5	5.0	266342.3	23.5

On average, it takes longer for a ROSE tool to repair a Defects4J error than a QuixBugs error. The repair time can be affected by a variety of factors such as the length of the execution from the starting point, the number of potential error lines, and the number of repairs that need to be validated. It varies with different errors.

It can be the case that ROSE does not suggest any repairs for an error, as all the potential repairs it generated may have a score lower than the validity score and were discarded. On average, it took ROSE-PC and ROSE-PS about 14.6 and 26.4 seconds on the average to report a no-repair-found result. The median time are only 3.9 and 10.6 seconds. This result suggests that ROSE fails fast for errors that it cannot repair.

> ROSE is highly efficient. A ROSE tool is quick at providing repair suggestions. It takes only seconds (often just a few seconds) for finding and reporting a correct repair. ROSE also fails fast for errors that it cannot repair.

Table 3. Results for Defects4J errors by ROSE-PC, the ROSE-based tool with the default pattern-based generators and ChatGPT used as repair suggesters. **#Results**: the number of repairs returned by the backend; **Correct Rank**: the rank of correct repair; **Total Time**: the total time for all repairs to be found; **Fix Time**: the time for the correct repair to be reported and viewed; **Fix Count**: the number of repairs validated; **SEEDE Count**: the total number of instructions SEEDE executed before processing the correct repair; **#Checked**: the number of repairs on which validation was run. All times reported are in seconds.

Defects4J Error Names / Statistics	#Results	Correct Rank	Total Time	Fix Time	Fix Count	SEEDE Count	#Checked
Chart01	3	1	27.9	8.7	14	52279	58
Chart09	17	3	35.2	4.5	6	3009	73
Chart11	4	1	15.5	2.2	2	1325	30
Chart12	2	1	22.5	1.6	4	8367	13
Chart20	1	1	14.1	3.5	10	1415	24
Chart14A	2	1	84.8	35.3	1	53	14
Chart17	4	1	14.5	3.0	3	258	13
Closure57	25	1	8.0	0.3	6	7948	48
Closure62	3	1	6.6	0.5	16	12291	32
Closure65	1	1	12.2	2.9	1	2698	81
Lang06	4	1	7.9	1.4	5	906	91
Lang21	1	1	4.6	0.4	2	11566	22
Lang24	12	1	8.4	0.7	2	344	142
Lang33	8	1	3.1	1.2	11	215	13
Lang39	32	3	6.8	1.4	9	1556	93
Lang58	3	1	11.4	3.4	37	7627	135
Lang59	5	1	4.0	0.8	10	271	23
Lang61	2	1	7.4	1.0	15	5740	100
Math02	24	1	38.0	8.9	24	512047	188
Math05	5	1	3.9	0.6	5	116	27
Math11	2	1	48.8	18.7	2	77126	23
Math22A	1	1	3.3	0.7	10	285	19
Math27	1	1	12.4	1.8	15	11665	108
Math30	2	2	18.4	10.6	21	372876	40
Math32	1	1	48.6	42.8	2	643	9
Math34	6	1	10.0	3.1	11	16556	23
Math41	14	1	11.4	1.0	11	15103	170
Math59	2	1	1.8	0.2	1	29	3
Math69	2	1	17.6	2.8	5	58758	72
Math75	5	2	5.6	2.4	21	38450	29
Math82	9	3	32.9	6.7	1	56140	18
Math94	4	1	5.4	0.5	1	234	32
Math96	5	1	7.0	1.1	9	850	54
Math105	47	1	11.1	0.9	1	163	93
Mockito38	5	1	3.0	0.3	4	432	10
Time04	33	2	52.7	12.4	38	110091	207
Time19	3	2	41.8	23.4	16	10777	46
Median	4.0	1.0	11.4	1.8	6.0	3009.0	32.0
Average	8.1	1.3	18.1	5.7	9.5	37843.5	58.8
StdDev	10.7	0.6	18.3	9.6	9.3	102525.4	53.5

6.1.4 ***Result for RQ3****: Number of Errors Repaired.* With the pattern-based and LLM-based repair suggesters, ROSE-PC found correct repairs for 36 QuixBugs and 37 Defects4J errors. Using the ChatGPT suggester, ROSE-PC repaired 26 Defecst4J errors and 30 QuixBugs errors. It used the pattern-based suggester to repair 11 Defects4J and 6 QuixBugs errors. Tables 2 and 3 list these errors together with other repair information including the number of repairs generated, the rank of correct repair, and the repair time. The result suggests that, a ROSE tool that uses advanced patch generators can repair many programming-assignment-level and real errors. With traditional suggesters, ROSE-PS found less correct repairs and addressed 14 QuixBugs and 19 Defects4J errors. Most (31, or 94%) of the errors were patched by the pattern-based suggester. Using the sharpFix suggester, ROSE-PS repaired two Defects4J errors. Table 4 shows the error names and the tool's repair result.

ROSE-PC found no repairs for 26 errors. For 15 (or 58%) errors, the failures were due to the inabilities of the underlying suggesters in making the correct repairs, not a problem of the ROSE framework per se. The other errors were not addressed due to the fault localization failure, as

Table 4. Results for QuixBugs and Defects4J errors by ROSE-PS, the ROSE-based tool with the default pattern-based generators and sharpFix-local. **#Results**: the number of repairs returned by the backend; **Correct Rank**: the rank of correct repair; **Total Time**: the total time for all repairs to be found; **Fix Time**: the time for the correct repair to be reported and viewed; **Fix Count**: the number of repairs validated; **SEEDE Count**: the total number of instructions SEEDE executed before processing the correct repair; **#Checked**: the number of repairs on which validation was run. All times reported are in seconds.

Benchmark	Error Names / Statistics	#Results	Correct Rank	Total Time	Fix Time	Fix Count	SEEDE Count	#Checked
QuixBugs	BitCount	5	1	0.7	0.1	1	18	25
	BreadthFirstSearch	4	1	4	0.7	3	320	17
	BucketSort	1	1	3.7	1.3	9	25190	36
	DetectCycle	5	1	2	0.2	2	56	25
	FindFirstInSorted	11	4	3	0.8	12	60333	50
	Gcd	2	1	1.5	0.1	2	72	15
	Hanoi	1	1	3.2	0.6	15	9515	43
	Knapsack	1	1	4.5	0.5	12	37846	67
	NextPermutation	1	1	4.5	0.1	2	1932	72
	Pascal	14	2	4.2	0.5	12	9432	70
	QuickSort	2	1	4.9	1	17	136326	75
	RpnEval	1	1	2.2	0.2	1	857	11
	Sieve	1	1	2.4	0.6	4	2487	21
	TopologicalOrdering	1	1	4.8	2	15	36981	31
	Median	1.5	1	3.4	0.5	6.5	5959.5	33.5
	Average	3.6	1.3	3.3	0.6	7.6	22954.6	39.9
	StdDev	4.1	0.8	1.3	0.5	6	37695.9	23
Defects4J	Chart01	1	1	26	9.8	25	73793	54
	Chart11	2	1	18.3	12.8	26	26395	51
	Chart12	1	1	30	10.5	5	10542	34
	Chart20	1	1	17.6	9.6	25	3870	37
	Chart14A	1	1	68	31.5	5	246	11
	Closure62	2	2	9	3.3	22	16760	132
	Closure65	2	1	11.5	2.5	1	2698	87
	Lang06	2	1	10.9	4.7	24	25228	82
	Lang21	1	1	9.3	1.1	3	11288	62
	Lang33	7	1	4.3	1.7	6	106	9
	Lang58	3	1	11.7	2.6	14	4025	114
	Lang59	2	1	4.6	1	10	273	20
	Math59	3	1	2.9	0.3	1	29	12
	Math75	7	1	14.2	10.1	31	85353	68
	Math94	3	1	6.6	0.6	1	234	36
	Math02	23	1	41.4	9.2	8	190435	173
	Math05	17	4	9.2	6.1	62	1663	114
	Math11	1	1	51.8	19	3	114581	19
	Mockito38	4	1	2.9	0.4	6	931	7
	Median	2	1	11.5	4.7	8	4025	51
	Average	4.4	1.2	18.4	7.2	14.6	29918.4	59.1
	StdDev	5.9	0.7	17.8	7.8	15.3	51105.9	47.3

discussed in Section 6.1.2. For 67 errors, ROSE-PS found no repairs, and its underlying suggesters did not produce correct repairs for 56 (or 84%) errors.

We also compared the ROSE tools with 28 existing APR techniques whose repairs were reported for the errors used in the experiment. We found that ROSE-PC correctly repaired more QuixBugs and Defects4J errors than all the techniques and that ROSE-PS repaired more QuixBugs errors than most techniques and more Defects4J errors than all but nine recent techniques including KNOD [41], AlphaRepair [123], and SelfAPR [137]. ROSE-PC and ROSE-PS are also faster than existing techniques. Detailed result can be found at [20]. We note that although we did a comparison because of ROSE's resemblance to existing APR techniques, the comparison is not direct, as the inputs, outputs, and goals of a ROSE tool differ from those of the program repair tools.

> A ROSE-based tool found correct repairs for as many as 36 of 40 QuixBugs and 37 of 60 Defects4J errors. A primary reason for the repair failures is the inability of the underlying repair suggesters in producing the correct repairs.

6.2 Utility Study

Table 5. Debugging tasks.

Task Id	Error Method Name	Error Benchmark	Error Root Cause	MLOC	Time Limit (min.)	ROSE gave correct suggestion
Task 1	gcd	Defects4J	Integer overflow	51	37.5	YES
Task 2	appendFixedWidthPadRight	Defects4J	Array Index Out Of Bounds	19	18.0	YES
Task 3	bitcount	QuixBugs	Logical Error	9	37.5	YES
Task 4	isValidParenthesization	QuixBugs	Logical Error	15	25.5	NO

We did a user study to investigate the usefulness and practicality of ROSE. For this study, we recruited 26 participants and assigned them to two groups. We chose four debugging tasks whose erroneous functionalities are not too project-specific and asked the participants to do the tasks with and without ROSE. We used the ROSE-PS tool, which does not find correct repairs for all tasks, for the study. By using this tool, we investigated both debugging scenarios where ROSE succeeds by finding a correct repair and fails with all incorrect repairs. We compared the performance of the two groups of participants in terms of success rate and debugging time. At the end of the study, we also did a survey in which we asked the participants to provide feedback about their experience of using ROSE and their feelings about ROSE's usefulness.

6.2.1 Study Design. In the study, participants first read a tutorial we created to learn how to use Code Bubbles [10] and ROSE, and then attempted four debugging tasks. If they use ROSE for a task t, we asked them to provide feedback regarding if and how ROSE helped in completing t right after t was done. At the end of the study, we did a survey in which we asked questions about their experience of using ROSE for debugging and their feelings about ROSE's usefulness.

The goal of the tutorial was to teach a participant how to use ROSE and Code Bubbles, the Eclipse-based IDE containing the ROSE user interface. In detail, it teaches a participant (a) how to view and edit code in Code Bubbles; (b) how to debug using the traditional IDE-based debugger without ROSE; and (c) how to debug using ROSE. To help the participant understand the details, we use an example to show, step by step, how to repair a Defects4J error Math59.

We designed four debugging tasks as shown in Table 5. From left to right, the table lists the task id (*Task Id*), the name of the error method (*Error Method Name*), the benchmark from which the error was chosen (*Error Benchmark*), the root cause of the error (*Error Root Cause*), the size of the error method in LOC (*MLOC*), the max time in minutes allowed to do the task (*Time Limit (min.)*), and whether ROSE provided correct repairs (*ROSE gave correct suggestion*).

For the first two tasks, we used the errors Math94 and Lang59 from the Defects4J benchmark; and for the last two, we used the errors BitCount and IsValidParenthesization from the QuixBugs benchmark. We note that ROSE was not successful on repairing all these errors. The error methods are designed to (1) compute the greatest common divisor of two input integers; (2) append an object to a string builder padding on the right to a fixed length; (3) count the number of bits of a given decimal; and (4) check whether the parentheses in a given string are matched. We chose these four errors because the functionalities of the error methods are relatively easy to understand, their fixes are not too complex nor too simple (according to a pilot study we did with 13 undergraduate students, the debugging time used to complete a task without using ROSE is ~19.7 minutes on average), and they cover the cases where ROSE can suggest both correct and incorrect repairs. Because the tasks are not equally difficult, we used different time limits, which we determined with the pilot study result. The time limit of a task is computed as $t \times 1.5$ where t is the average time taken to complete the task without ROSE in the pilot study and 1.5 is used to account for more flexibility.

For each task, we showed the participant the failure along with the input triggering the failure. To make sure debugging could be done in a reasonable amount of time, we additionally informed

the participant the error method where fixes should be made and the functionality of the error method along with its expected behavior for the given input. Note that we did not restrict ROSE's focus to the error method to make its repair process easier. For participants who were allowed to use ROSE, we did not tell them where to set the stopping point to invoke ROSE and whether ROSE could give correct suggestions or not. For each task, we recorded the time a participant used to complete it. To decide whether a participant succeeded for a task, we manually inspected the repair made by the participant and compared it against the developer repair provided by the benchmark.

We recruited 26 participants who were all students from Wuhan University. Four of the participants were graduate students and the others were undergraduates. All participants have at least two years' programming experience and know how to write code in Java. To implement a controlled experiment, we randomly assigned a participant to one of two groups, either Group A or Group B. Each group has 13 participants. We asked participants in Group A to complete Task 1 and Task 2 with ROSE and the other two without ROSE. We asked participants in Group B to do the opposite, i.e., complete Task 1 and Task 2 without ROSE and the others with ROSE.

After each task, we asked participants who used ROSE in the task to provide feedback regarding if and how ROSE helped. At the end of the study, participants are asked to fill in a questionnaire to answer eight questions including five single-choice questions and three free-answer questions to provide their feedback regarding the experience of using ROSE and ROSE's usefulness. The second column of Table 7 presents the five single-choice questions. The three free-answer questions are (1) if ROSE was helpful, how did it help? (2) if ROSE was not great, what made you feel unwilling to use ROSE in the future? and (3) how to improve ROSE?

Table 6. Result of the debugging tasks.

Task Id	Without ROSE		With ROSE	
	Success # (%)	Time (min.)	Success # (%)	Time (min.)
Task 1	5 (38.5)	23.7	11 (84.6)	14.0
Task 2	10 (76.9)	12.6	13 (100)	8.1
Task 3	9 (69.2)	18.6	13 (100)	7.7
Task 4	9 (69.2)	13.5	11 (84.6)	13.6
Average	**8.3 (63.5)**	**17.1**	**12 (92.3)**	**10.9**

6.2.2 ***Result for RQ4****: ROSE's usefulness and practicality.* Table 6 details how the participants performed for the debugging tasks with and without using ROSE in terms of the success number and rate (*Success # (%)*) and the debugging time (*Time (min.)*). The success number denotes the number of participants who successfully completed the debugging task and repaired the error correctly within the time limit. The success rate denotes the percentage of participants who succeeded in the task. Debugging time is the average time used by participants who successfully completed the task.

On average, 92.3% of participants who used ROSE succeeded in the debugging task whereas only 63.5% of participants who did not use ROSE succeeded. This shows ROSE helped 44.6% more participants find the correct repair for a debugging task. Moreover, participants who used ROSE and succeeded spent 10.9 minutes to complete the debugging task, whereas participants who did not use ROSE spent 17.1 minutes (56.9% more time) to make a correct repair. This shows ROSE can help reduce debugging time.

We used the paired t-test method [60] to investigate the significance of performance difference. This result shows the differences of success number and rate are significant (p-values are both 0.022), which implies ROSE significantly helped more participants succeed in the debugging tasks. The difference of debugging time is overall insignificant (p-value is 0.09). As Table 6 shows, for

Task 4, ROSE did not generate correct repairs, and participants who used ROSE spent slightly more time for debugging than those who did not use ROSE. For this task, ROSE generated 18 repair suggestions, which are all incorrect. It is possible that participants who used ROSE in this case checked many of these suggestions one by one by previewing what has been changed, applying the repair to the program, executing the repaired program, and analyzing the result. An incorrect repair can be misleading and slow down the debugging process. Nevertheless, according to the survey result, 53.8% of participants thought that ROSE was useful in this case because the suggested repairs, albeit incorrect, helped them identify the faulty variables and statements. For example, a participants who used ROSE for Task 4 said: "Maybe the suggestions are not so correct, but it gives directions to figure out how this bug appears".

Table 7. Result of the five survey questions.

Question Id	Question	Score
Q1	You find ROSE useful to help you with your tasks.	4.5
Q2	What do you think of the idea of ROSE?	4.2
Q3	ROSE would be useful for debugging tasks in general.	4.2
Q4	How do you like ROSE in general?	4.2
Q5	Would you consider using ROSE to debug your own code?	3.5

Table 7 presents the single-answer questions, which were designed to understand the participants' experience of using ROSE and their feelings about ROSE's usefulness. For each question, the participant gives a score from 1 and 5, where 1 represents the most negative feedback and 5 represents the most positive feedback. For example, 5 for first question means that the participant strongly feels that ROSE was useful while 1 means the participant thinks ROSE was not useful at all.

The average scores for the first four questions are all above 4. This shows overall the participants find ROSE useful for debugging and they like ROSE. The score for Q5 is 3.5. This implies their willingness of using ROSE to debug their own code is not so strong. Based on their detailed feedback, we realized that this is mostly related to the design of Code Bubbles (the IDE where ROSE is currently integrated), the GUI interface, presentation, and aesthetics. For example, some of the participants complained that the shortcut keys (for viewing and editing code) are not typical, the fonts are sometimes small, and colors are not what they like. Many participants also mentioned that they are more willing to use ROSE if it could be integrated into other IDEs such as IntelliJ IDEA [35] and Visual Studio Code [111]. We believe these can all be further improved.

> ROSE helped 44.6% more participants succeed in the debugging tasks and helped reduce the debugging time by 36.2%. Participants found ROSE useful and they liked ROSE.

6.3 Threats to Validity

Our evaluation shows that ROSE is a very effective and practical framework that supports quick repair of semantic errors and is helpful for debugging. There are however both external and internal threats that can potentially harm the validity of the evaluation. As for external threats, our experiment conducted to investigate the effectiveness and usefulness of ROSE is based on repair tools configured with two sets of repair suggesters. We used errors from two benchmarks for evaluation, and the user study includes a limited number of participants doing four debugging tasks. The experiment result may not generalize to other suggesters and errors. The study result could also vary with different participants and tasks.

We nevertheless note that we included both advanced (including ChatGPT) and traditional techniques, and they were selected because they are both efficient and accurate and are thus well-suited for ROSE's quick-repair goal. We focused on investigating how ROSE's key components (i.e., fault localization and patch validation) work with these patch generators configured, and we found the two components were both very effective. The errors used in our experiment have been used to evaluate other existing repair techniques. Our user study, albeit limited, is non-trivial and is comparable to existing debugging-based studies (such as [63] and [98]) in terms of the number of participants and tasks.

To mitigate internal threats, we thoroughly tested the implemented code of ROSE and carefully checked the repair results to gain confidence in the validity of our evaluation. We note that the current implementation of ROSE is a prototype. We plan to investigate further improving it by for example handling Java reflection.

As for threats to ROSE's utility, it is possible that if the developers can specify the symptom, it would not be too difficult to manually repair the error. While this is possible, it is not typical. In other words, we think it is usually the case that a developer knows a problem caused by the error, but this does not necessarily make debugging easier, because the location where a problem arises may not be where changes should be made (which is why fault localization techniques exist) and even if the developer knows where to change, it does not necessarily mean that finding a valid repair in only a few seconds would be easy. A ROSE-based tool, which works in seconds to make repair suggestions, can offer help. Our study results support this.

Another threat to utility relates to the participants of the user study we did to evaluate ROSE. The participants are all students. Although they have at least two years' programming experience to be eligible for participation, they are not professional software engineers, and their results may not reflect how useful ROSE is to professional software developers. The study can however be extended with professional software engineers involved to investigate the utility of ROSE for highly skilled developers.

ROSE is a repair framework that leverages the power of existing APR patch generators to achieve quick repair of semantic errors and facilitate debugging. Its current focus is on dealing with relatively simple errors. We experimented with single-line errors. But it is possible to also use ROSE to handle single-hunk, multi-line errors provided that the patch generators plugged in ROSE can generate multi-line patches. Because ROSE's fault localization assumes that each potential location be treated independently, it does not support repairing multi-hunk errors whose fix changes multiple locations of the program. We also note that ROSE can be less helpful when applied to handling complex errors that require multi-line or even multi-hunk changes for fixing. Nevertheless, as we previously showed, a ROSE-based tool fails fast when it generates no repairs (Section 6.1.3) and even when it generates incorrect repairs, these may still provide useful repair hints and do not considerably slow down the debugging time (Section 6.2). We also note that it is possible to integrate more advanced patch classification and ranking techniques (such as [27]) to further reduce the chance of reporting incorrect repairs, which we leave as future work.

The effectiveness of a ROSE-based technique depends on all of its components: fault localization, patch generation, and patch validation. These components have dependencies. The output of a previous component is the input of the following. In this way, any inaccuracy of a component can affect its following components, and errors could accumulate and finally lead to ineffective repair. This is a known problem for APR in general. We note that ROSE's approach can mitigate the accumulation to some extent, as its baseline execution generation can filter out potential faulty lines that are identified by fault localization but do not actually execute and its validation can also exclude candidate patches that do not exhibit any failure-resolving behaviors.

ROSE's patch validation is based on simulated trace comparison. The comparison uses heuristics and scores for patch prioritization. Although Section 6.1.2 shows that the validation is empirically effective, there can be cases where the heuristics and scores do not lead to a good prioritization. For example, the heuristic that ROSE uses for validating exception or assertion problems gives high priority to a repaired execution that avoids the error and is close to the original baseline execution. However, it can be the case that a correctly repaired execution is not highly close to the original baseline (consider a repaired execution that avoids the exception with an early return) and is thus not given a high rank. In future work, we plan to explore learning-based approaches for improvement.

7 Conclusion and Future Work

Due to the unrealistic assumptions and the low-efficiency characteristic, automated program repair (APR), though promising to deal with semantic errors, has not been shown to be practical while debugging. To improve the practicality of APR and make it an everyday part of IDE to help people debug, we developed ROSE, a framework that allows the integration of existing APR patch generators to produce quick-repair suggestions for semantic errors. To ensure practicality and usefulness, ROSE does not require a test suite or program re-execution for problem specification and patch validation. Rather, ROSE interacts with the developer to obtain a problem symptom describing the error. It then performs test-free fault localization accounting for the program states and their dependency while working closely with the debugger to identify potential repair locations. After generating a baseline execution and invoking the patch generators to propose repairs for these locations, ROSE validates the repairs via simulating and comparing traces while accounting for various matching conditions and the problem symptom to suggest repairs likely to be correct.

The effectiveness and utility studies used to evaluate the efficacy and usefulness of ROSE show that ROSE's repair can provide correct suggestions for many semantic errors in seconds, that ROSE's test-free fault localization and patch validation are highly effective, and that ROSE can indeed help developers debug. A video showing ROSE in action can be seen at https://youtu.be/GqyTPUsqs2o.

In future work, in addition to improving ROSE to handle complex language features (such as Java reflection), incorporating advanced static patch assessment approaches (such as [27, 108, 135]), and performing an extensive experiment with more APR patch generators used for repair and more diverse errors for evaluation, we will explore enhancing ROSE's ability to address complex multi-hunk errors, which require changes of multiple locations for repair. One possible direction for tackling multi-hunk errors would be performing iterative repair by localizing and proposing single-location patches and evolving them into multi-location repairs. We will explore leveraging both learning-based approaches and human feedback to provide critical guidance. ROSE's presentation highlights the code changes made in a repair but does not provide in-depth analyses showing why the changes can fix the problem. It would be interesting to explore generating for each repair an explanation in natural language (via for example LLMs) showing more debugging insights. The implementation of ROSE is currently tied to the Code Bubbles IDE supporting Java language. It can however be refactored to provide APIs adhering to the Language Server Protocol (LSP) and the Debug Adapter Protocol (DAP) to allow easy integration into other IDEs and platforms. We will continue to improve ROSE (and the open source SEEDE and FAIT tools it uses). The open-source code base of the latest version of ROSE can be found at https://github.com/StevenReiss/rose. The artifact of this work is available at https://github.com/rose-apr/rose.

Acknowledgments

We are grateful for the valuable comments and suggestions given by the anonymous reviewers. This work was partially supported by the National Natural Science Foundation of China under the grant numbers 62202344 and 62141221 and the OPPO Research Fund.

References

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* (2009), 1780–1792.
[2] Afsoon Afzal, Manish Motwani, Kathryn T Stolee, Yuriy Brun, and Claire Le Goues. 2019. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2162–2181.
[3] Hiralal Agrawal, Richard A DeMillo, and Eugene H Spafford. 1993. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience* 23, 6 (1993), 589–616.
[4] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6 (1990), 246–256.
[5] Auto Correct 2023. *Visual Studio's Auto Correct.* https://marketplace.visualstudio.com/items?itemName=sygene.auto-correct
[6] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.
[7] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. 2019. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering*. 613–624.
[8] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint Meeting on the Foundations of Software Engineering*. 179–190.
[9] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. 2020. Human-in-the-loop automatic program repair. In *Proceedings of IEEE 13th International Conference on Software Testing, Validation and Verification*. 274–285.
[10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. 455–464.
[11] Renée C Bryce, Alison Cooley, Amy Hansen, and Nare Hayrapetyan. 2010. A one year empirical study of student programming bugs. In *Proceedings of IEEE Frontiers in Education Conference*. F1G–1.
[12] Eduardo Cunha Campos and Marcelo de Almeida Maia. 2017. Common bug-fix patterns: A large-scale observational study. In *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 404–413.
[13] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. *arXiv preprint arXiv:2304.08191* (2023).
[14] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.
[15] ChatGPT 2023. *ChatGPT.* https://chat.openai.com/
[16] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and precise on-the-fly patch validation for all. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering*. 1123–1134.
[17] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.
[18] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.
[19] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*. 342–351.
[20] d4j28toolresult 2022. *Defects4J Errors Repaired by 28 APR techniques.* https://docs.google.com/spreadsheets/d/1uo5mVTZPRYx0oLrkll1N4Ab3hazcAMWvPw2lH7NC2gg/edit?usp=sharing
[21] Higor A de Souza, Marcos L Chaim, and Fabio Kon. 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347* (2016).
[22] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.
[23] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering. 935–947.
[24] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 8–18.
[25] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology* 30, 2 (2021), 1–27.
[26] Ali Ghanbari and Andrian Marcus. 2020. PRF: a framework for building automatic program repair prototypes for JVM-based languages. In *Proceedings of the 28th ACM Joint Meeting on the Foundations of Software Engineering*. 1626–1629.
[27] Ali Ghanbari and Andrian Marcus. 2022. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 654–665.
[28] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based program repair using SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 173–188.
[29] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* (2019), 56–65.
[30] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d'Amorim, and Thomas Reps. 2021. Shipwright: A Human-in-the-Loop System for Dockerfile Repair. In *Proccedings of IEEE/ACM 43rd International Conference on Software Engineering*. 1148–1160.
[31] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 464–475.
[32] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-tuning Large Language Models of Code for Automated Program Repair. (2023).
[33] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 539–554.
[34] Infer 2023. *Infer*. https://fbinfer.com/
[35] IntelliJ IDEA 2022. *IntelliJ IDEA*. https://www.jetbrains.com/idea/
[36] Werner Janjic, Oliver Hummel, Marcus Schumacher, and Colin Atkinson. 2013. An unabridged source code dataset for research in software reuse. In *Proceedings of 10th Working Conference on Mining Software Repositories*. 339–342.
[37] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. 2009. BugFix: A learning-based tool to assist developers in fixing bugs. In *Proceedings of 17th International Conference on Program Comprehension*. 70–79.
[38] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault localization using value replacement. In *Proceedings of the International Symposium on Software Testing and Analysis*. 167–178.
[39] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM International Symposium on Software Testing and Analysis*. 298–309.
[40] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020* (2023).
[41] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain knowledge distilled tree decoder for automated program repair. In *Proccedings of 43th International Conference on Software Engineering (to appear)*.
[42] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering*. 1161–1173.
[43] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).
[44] Wei Jin and Alessandro Orso. 2012. Bugredux: Reproducing field failures for in-house debugging. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 474–484.
[45] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.
[46] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. MintHint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*. 266–276.
[47] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2023. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging. *arXiv preprint arXiv:2304.02195* (2023).
[48] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE,

2312–2323.
[49] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 295–306.
[50] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.
[51] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, et al. 2021. On the introduction of automatic program repair in Bloomberg. *IEEE Software* 38, 4 (2021), 43–51.
[52] Andrew J Ko and Brad A Myers. 2004. Designing the Whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.
[53] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. 2013. An empirical study of adoption of software testing in open source projects. In *Proceedings of 13th International Conference on Quality Software*. 103–112.
[54] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* (2020), 1980–2024.
[55] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug report driven program repair. In *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering*. 314–325.
[56] Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 579–590.
[57] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting on the Foundations of Software Engineering*. 593–604.
[58] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE.
[59] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. GenProg: A generic method for automatic software repair. *IEEE transactions on software engineering* (2011), 54–72.
[60] Erich Leo Lehmann, Joseph P Romano, and George Casella. 2005. *Testing statistical hypotheses*. Vol. 3. Springer.
[61] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 169–180.
[62] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *Proccedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673.
[63] Jingjing Liang, Ruyi Ji, Jiajun Jiang, Shurui Zhou, Yiling Lou, Yingfei Xiong, and Gang Huang. 2021. Interactive Patch Filtering as Debugging Aid. In *Proceedings of 37th IEEE International Conference on Software Maintenance and Evolution*. 239–250.
[64] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 55–56.
[65] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 393–403.
[66] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically generating bug fixes from bug reports. In *Proceedings of IEEE 6th international conference on software testing, verification and validation (ICST)*. IEEE, 282–291.
[67] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.
[68] Yu Liu, Sergey Mechtaev, Pavle Subotić, and Abhik Roychoudhury. 2023. Program Repair Guided by Datalog-Defined Static Analysis. In *Proceedings of the 31st ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE) to appear*.
[69] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.
[70] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on the Foundations of Software Engineering*. 166–178.
[71] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 298–312.

[72] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM International Symposium on Software Testing and Analysis*. 75–87.

[73] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[74] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*. 101–114.

[75] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*. 94–105.

[76] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.

[77] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering* 22, 4 (2017), 1936–1964.

[78] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 129–139.

[79] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. IEEE, 448–458.

[80] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. 691–701.

[81] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*. 1169–1180.

[82] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2023. Template-based Neural Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1456–1468.

[83] Martin Monperrus. 2018. Automatic software repair: a bibliography. *Comput. Surveys* 51, 1 (2018), 1–24.

[84] Martin Monperrus. 2020. The Living Review on Automated Program Repair. (Dec. 2020). https://hal.archives-ouvertes.fr/hal-01956501 working paper or preprint.

[85] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Proccedings of Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[86] Manish Motwani and Yuriy Brun. 2023. Better automatic program repair by using bug reports and tests together. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1225–1237.

[87] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[88] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[89] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the 44th International Conference on Software Engineering*. 2228–2240.

[90] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315.

[91] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* (2015), 605–628.

[92] Yu Pei, Carlo A Furia, Martin Nordio, and Bertrand Meyer. 2015. Automated program repair in an integrated development environment. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 681–684.

[93] Yu Pei, Carlo A Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated fixing of programs with contracts. *Ieee transactions on software engineering* 40, 5 (2014), 427–449.

[94] Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael R Lyu. 2023. Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors. *arXiv preprint arXiv:2306.01394 (to appear at ICSE'24)* (2023).

[95] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.
[96] Quick Fix 2023. *Eclipse's Quick Fix*. https://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F
[97] Steven P. Reiss. 2019. Continuous Flow Analysis to Detect Security Problems. *arXiv preprint arXiv:1909.13683* (2019).
[98] Steven P. Reiss, Qi Xin, and Jeff Huang. 2018. SEEDE: simultaneous execution and editing in a development environment. In *Proceedings of 33rd IEEE/ACM International Conference on Automated Software Engineering*. 270–281.
[99] Tobias Roehm, Nigar Gurbanova, Bernd Bruegge, Christophe Joubert, and Walid Maalej. 2013. Monitoring user interactions for supporting failure reproduction. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 73–82.
[100] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 345–355.
[101] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. 2019. Harnessing evolution for multi-hunk program repair. In *Proceedings of IEEE/ACM 41st International Conference on Software Engineering*. 13–24.
[102] Kostyantyn M Shchekotykhin, Thomas Schmitz, and Dietmar Jannach. 2016. Efficient Sequential Model-Based Fault-Localization with Partial Diagnoses.. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. 1251–1257.
[103] André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *arXiv preprint arXiv:2312.15698* (2023).
[104] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 532–543.
[105] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).
[106] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2018. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1294–1317.
[107] Shin Hwei Tan, Ziqiang Li, and Lu Yan. 2024. CrossFix: Resolution of GitHub issues via similar bugs recommendation. *Journal of Software: Evolution and Process* 36, 4 (2024), e2554.
[108] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. 2023. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–34.
[109] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *Proceedings of 37th IEEE International Conference on Software Engineering*. 403–414.
[110] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*. 151–162.
[111] Visual Studio Code 2022. *Visual Studio Code*. https://code.visualstudio.com/
[112] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 international symposium on software testing and analysis*. 1–11.
[113] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.
[114] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*. 61–72.
[115] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.
[116] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 356–366.
[117] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2348–2368.
[118] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of IEEE/ACM 40th International Conference on Software Engineering*. 1–11.

[119] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on the Foundations of Software Engineering*. 354–366.

[120] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[121] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. *arXiv preprint arXiv:2303.10494 (to appear at ICSE'24)* (2023).

[122] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[123] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint Meeting on the Foundations of Software Engineering*. 959–971.

[124] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[125] Yuan-An Xiao, Chenyang Yang, Bo Wang, and Yingfei Xiong. 2023. ExpressAPR: Efficient Patch Validation for Java Automated Program Repair Systems. In *Proceedings of 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Demonstration Track*.

[126] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on software engineering and methodology (TOSEM)* 22, 4 (2013), 1–40.

[127] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering*. 660–670.

[128] Qi Xin and Steven P. Reiss. 2019. Better code search and reuse for better program repair. In *2019 IEEE/ACM International Workshop on Genetic Improvement*. 10–17.

[129] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. 789–799.

[130] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of IEEE/ACM 39th International Conference on Software Engineering*. 416–426.

[131] Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu, and Baowen Xu. 2018. Debugging with intelligence via probabilistic inference. In *Proceedings of the 40th International Conference on Software Engineering*. 1171–1181.

[132] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[133] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2022. TransplantFix: Graph Differencing-based Code Transplantation for Automated Program Repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[134] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024).

[135] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering* 48, 8 (2021), 2920–2930.

[136] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021), 110825.

[137] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[138] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*. 1506–1518.

[139] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. 2022. CIRCLE: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 678–690.

[140] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on software engineering (TSE)* 46, 10 (2018), 1040–1067.

[141] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward better evolutionary program repair: An integrated approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–53.

[142] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.
[143] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann.
[144] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *arXiv preprint arXiv:2301.03270* (2023).
[145] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. 2023. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879* (2023).
[146] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*. 272–281.
[147] Xindong Zhang, Chenguang Zhu, Yi Li, Jianmei Guo, Lihua Liu, and Haobo Gu. 2020. Precfix: Large-scale patch recommendation by mining defect-patch pairs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 41–50.
[148] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 691–702.
[149] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427* (2024).
[150] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *Proceedings of 34th International Conference on Software Engineering*. 14–24.
[151] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on the Foundations of Software Engineering*. 341–353.