Explainable Automated Debugging via Large Language Model-driven Scientific Debugging

Sungmin Kang*
sungmin.kang@kaist.ac.kr
KAIST
Daejeon, South Korea

Bei Chen
bei.chen@microsoft.com
Microsoft Research Asia
Beijing, China

Shin Yoo
shin.yoo@kaist.ac.kr
KAIST
Daejeon, South Korea

Jian-Guang Lou
jlou@microsoft.com
Microsoft Research Asia
Beijing, China

ABSTRACT

Automated debugging techniques have the potential to reduce developer effort in debugging, and have matured enough to be adopted by industry. However, one critical issue with existing techniques is that, while developers want rationales for the provided automatic debugging results, existing techniques are ill-suited to provide them, as their deduction process differs significantly from that of human developers. Inspired by the way developers interact with code when debugging, we propose Automated Scientific Debugging (AutoSD), a technique that given buggy code and a bug-revealing test, prompts large language models to automatically generate hypotheses, uses debuggers to actively interact with buggy code, and thus automatically reach conclusions prior to patch generation. By aligning the reasoning of automated debugging more closely with that of human developers, we aim to produce intelligible explanations of how a specific patch has been generated, with the hope that the explanation will lead to more efficient and accurate developer decisions. Our empirical analysis on three program repair benchmarks shows that AutoSD performs competitively with other program repair baselines, and that it can indicate when it is confident in its results. Furthermore, we perform a human study with 20 participants, including six professional developers, to evaluate the utility of explanations from AutoSD. Participants with access to explanations could judge patch correctness in roughly the same time as those without, but their accuracy improved for five out of six real-world bugs studied: 70% of participants answered that they wanted explanations when using repair tools, while 55% answered that they were satisfied with the Scientific Debugging presentation.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Automated Program Repair, Machine Learning

1 INTRODUCTION

Automated debugging techniques, such as Fault Localization (FL) or Automated Program Repair (APR), aim to help developers by automating the debugging process in part. Due to the significant amount of developer effort that goes into debugging [41], automated debugging is a research topic of significant interest [20]: many papers are published every year [25], and the field is mature enough to see adoption by industry [16, 21].

Regarding the practical adoption of these techniques, a body of literature surveying developer expectations on automated debugging has consistently highlighted that, as much as strong performance on software engineering tasks is important, so is supporting information that helps developers judge the results. For example, Kochhar et al. [17] perform a study of developer expectations on fault localization, and find that more than 85% of developers agree that the ability to provide rationale is important. Further, Kirbas et al. [16] note that some developers responded negatively to automated program repair results, citing that they would come "out of the blue". Such findings suggest that strong automated debugging results may not be acceptable on their own, and may need supporting information that helps *explain* the results.

Despite the consistent request for explainable processes for automated results, to the best of our knowledge explainable automated debugging techniques can be difficult to come by. For example, in the living review of APR compiled by Monperrus updated in August 2022 [25], the word 'explain' appears only in one position paper [24], revealing that the critical research on how to explain repair suggestions to developers is under-explored. We argue that this is in part because existing automated debugging techniques reason in starkly different ways to humans. Whereas existing automated debugging techniques will reduce a search space [10] and try multiple solutions to find results that are correlated with the location and fix of a bug [26], human developers will generally utilize debuggers and `print` statements to interact with the buggy code, understand its behavior and in turn make a patch based on such observations [32]. That is, the reasoning traces [19] of existing automated debugging processes are so different from those of developers, that suggesting them may contribute little to the understanding of a generated patch.

As a step towards automated debugging techniques that can generate explanations that help developers, we propose AutoSD, which bridges the gap between humans and automated debugging processes. To do so, AutoSD leverages Large Language Models (LLMs) and a debugger interface to automatically emulate the Scientific Debugging (SD) process for developers proposed by Zeller [41]. AutoSD prompts an LLM to automatically generate hypotheses about

*This work was done as part of an internship at Microsoft Research Asia.

what is causing the bug, along with a debugger script that would test the hypotheses. AutoSD then executes the suggested debugger command and provides the LLM with the result; based on this, the LLM finally decides whether the hypothesis was met, and predicts if the debugging process is done, or additional investigation is required. The intermediate debugging text generated as a result can naturally be presented as an *explanation* describing how AutoSD reached its conclusion. Emulating Scientific Debugging has ideal properties for explainable debugging: notably, as existing work identifies that developers use the principles of Scientific Debugging to debug even without formal training [32], the explanations could help inform or augment the thought process of developers.

We empirically evaluate AutoSD by first evaluating it on three program repair benchmarks. Our results indicate AutoSD can achieve competitive repair results to non-explainable APR techniques. In terms of practical usage, precision is an important factor [39]; we find that for cases when AutoSD indicates it had collected enough information for debugging, repair performance is in fact higher. As language models become more capable, the repair performance of AutoSD rapidly increases as well, demonstrating the potential of AutoSD. We further perform a user study on Python developers involving 20 participants, including six professional developers, under a realistic APR application setting: reviewing patches for acceptance. Our results demonstrate that the debugging traces generated by AutoSD enhance developer accuracy in terms of accessing whether the patch is correct for 83% of the real-world bugs studied, while keeping the amount of time in which developers could judge whether the patch roughly constant; these results suggest that humans benefit from the automatically generated patch explanations of AutoSD. Furthermore, 70% of participants responded that they would see explanations as an important factor when using APR tools, and 55% were satisfied with the Scientific Debugging formulation of AutoSD.

Overall, our contribution may be summarized as:

- We identify that explainable automated debugging may be achieved by LLMs emulating developer processes, and as a demonstration propose AutoSD, which uses LLMs to emulate Scientific Debugging [41];
- We perform empirical analyses on three APR benchmarks, demonstrating that AutoSD can achieve significant APR performance while also generating explanations;
- We conduct a developer study on AutoSD, based on a realistic scenario of patch review, and demonstrate explanations from AutoSD can aid developers in decision-making;
- We further solicit feedback from users regarding repair explanations, presenting a guideline for future improvement of explanations.

The remainder of the paper is organized as follows. We introduce the technical background to our work in Section 2, and our technique AutoSD in Section 3. The evaluation setup and research questions are provided in Section 4, and the empirical results based on these experiments are presented in Section 5. Threats and limitations are discussed in Section 6, and Section 7 concludes.

2 BACKGROUND

This section provides the motivation and background for our work.

2.1 Explainable Automated Debugging

Automated debugging has a long history, with research often being done on the topics of fault localization [14, 18, 26] and automated program repair [7]. As described before, while the technical complexity and performance of automated debugging techniques has been increasing [12], including the use of LLMs for APR [11, 37], empirical work on explaining results for developer consumption has been difficult to identify. In addition to Monperrus' living review on APR having only one paper mentioning explanations [25], Winter et al. [36] find 17 human studies evaluating APR, of which none involved explanations directly from an APR tool; Kochhar et al. [17] survey fault localization techniques at the time, and find two techniques that could provide explanations of their results [22, 33]; unfortunately, both papers did not have human studies.

This contrasts to the growing body of literature showing that, to adopt automated debugging techniques in practice, 'explanations' for the results would be welcome. Developers have stated their desire for explanations in multiple occasions: along with the findings of Kochhar et al. [17] mentioned earlier, a developer study on expectations for APR by Noller et al. [28] notes that "the most commonly mentioned helpful output from an APR tool is an *explanation* ... including its *root cause*". Developer expectation is particularly important because when automated debugging has been adopted by industry, automatically generated patches are consistently reviewed by developers. At Meta, the APR system is connected to the internal code review platform [21]; at Bloomberg, Kirbas et al. [16] write that "Bloomberg's view was that full automation was far from ideal", and they subject APR patches to be reviewed by a software engineer. This is also reflected in Noller et al.'s results that "full developer trust requires a manual patch review".

A promising way to present developers with explanations could be to show the reasoning trace [19] of a tool, i.e. how an automated debugging tool came to recommend a certain line for FL or a certain patch for APR. Unlike post-hoc explanation techniques such as commit message generation [13], such reasoning traces can answer critical questions that a developer may have, such as 'why this patch?'; indeed, research in Human-Computer Interactions (HCI) have indicated that explanations should strive to be capable of answering *why* an approach gave a certain result [19].

However, current automated debugging techniques are ill-suited to generate helpful explanations for their results, as their reasoning traces deviate from human reasoning traces significantly. Using a common classification of APR techniques [8] as an example[1], generate-and-validate (G&V) techniques [7] (which includes learning-based techniques [10, 38, 42]) will generate variants of the buggy code until a test passes. As their deduction process is simply enumerating changes and trying them one by one, the process runs without regard to any 'root cause'. Semantics-based APR techniques such as Angelix [23] use variable values as inputs to Satisfiability Modulo Theory (SMT) solvers to more effectively search within a patch space; thus they are not inherently identifying any 'root cause' either. This is not to say these techniques are ineffective at fixing bugs - numerous work on APR shows that existing APR techniques can fix a wide array of bugs. Rather, we argue that because their reasoning trace is so different from humans, it is difficult to

[1]The explainability of FL is discussed in the appendix.

make a satisfactory explanation of their results. On the other hand, one way to make satisfactory explanations would be to develop an automated debugging technique that deduces in a similar way to humans, to make the decision-making process transparent [4].

2.2 Scientific Debugging

To align APR reasoning traces more closely to those of human developers, we must know how developers debug in practice. Previous work on developer debugging patterns provide glimpses into how debugging is actually done.

Early work on developer debugging found that there was a "gross descriptive model" that developers followed, in which developers formulated hypotheses, then verified whether the hypotheses are true [9]. A formal version of this process was named *Scientific Debugging* by Zeller [41], who advocated for developers to maintain a debugging log consisting of an iteration of the following items:

- Hypothesis: a tentative description that explains the bug and is consistent with the known observations;
- Prediction: an expected outcome if the hypothesis is true;
- Experiment: a means of verifying the prediction;
- Observation: the result of an experiment;
- Conclusion: a judgement of the hypothesis, based on the observation.

Siegmund et al. [32] found that even without formal training in debugging techniques, all developers surveyed would roughly follow the 'hypothesis formulation, then verification' process of scientific debugging. Thus, Scientific Debugging can be seen as a formal way of describing the dominant developer thought process when debugging, and thus we seek to emulate this process to make an explanation when generating APR results.

2.3 Large Language Models

In this paper, we seek to emulate the Scientific Debugging process via Large Language Models (LLMs). We believe LLMs are capable of emulating Scientific Debugging for the following reasons. First, they have shown increasingly strong performance on question-answering benchmarks that involve reasoning [2, 29], which also makes it possible that they would be capable of predicting whether a hypothesis is met, and which hypothesis to investigate next. While it would be difficult to manually gather a large amount of data that contains debugging traces in the Scientific Debugging format, LLMs have also been demonstrated to be capable of few-shot or zero-shot problem solving: that is, given a few examples or simply a description of the task to be solved in the form of a natural-language *prompt*, they are capable of doing the task [2]. This capability improves with Reinforcement Learning with Human Feedback (RLHF) training [30], which the main LLM of our task (ChatGPT of OpenAI) was trained on. Finally, the interaction with code that Scientific Debugging asks for requires the use of external tools. When using 'Chain-of-Thought' (CoT) prompting [34], LLMs appear capable of using the results of external tools to improve their performance as well [6, 40]. As a result, we believe that LLMs are well-positioned to emulate the Scientific Debugging process, and thus generate reasoning traces intelligible to developers.

3 AUTOMATED SCIENTIFIC DEBUGGING

The overall process of our approach is presented in Figure 1. To start, the prompt containing relevant information is generated (Figure 1 Ⓐ): this consists of a detailed explanation of what Scientific Debugging is, and a description of the debugging problem itself, so that AutoSD can proceed with the following steps. With the initial prompt prepared, AutoSD generates a hypothesis on what is wrong with the code or how it can be fixed, along with the concrete experiment that would validate such a hypothesis, using an LLM (Figure 1 Ⓑ). The experiment script will be passed to a background debugger/code executor process, which runs the script and returns the actual result (Figure 1 Ⓒ). Based on the observed information, AutoSD decides whether the hypothesis was verified or not using an LLM (Figure 1 Ⓓ); depending on the conclusion, AutoSD either starts with a new hypothesis or opts to terminate the debugging process and generate a fix. When the interaction with the code is over, AutoSD generates a bug fix based on the gathered information (Figure 1 Ⓔ). Unlike other automated program repair techniques we are aware of, as a result of steps (Ⓑ - Ⓓ) AutoSD can provide a *rationale* of how a particular fix was generated, which can then be provided to the developer upon request.

3.1 Constructing the Input Prompt

To construct the initial prompt, as in the example presented in Figure 1 Ⓐ, we first manually wrote a detailed description of Scientific Debugging that explains what hypotheses, predictions, experiments, observations, and conclusions are, along with multiple examples for each category, so that the LLM can generate an intelligible reasoning trace. The full description can be found in the appendix; here, we describe the aspects of the description critical for the pipeline of AutoSD in detail. For one, concrete examples of experiments are provided, to allow the LLM to predict appropriate experiment scripts: composite debugger commands (consisting of setting a breakpoint, running code, and printing a value) and a Domain-Specific Language (DSL) that we define to allow edit-and-execute commands are given. Regarding the DSL, the prompt specifies that the following commands are available: `REPLACE(line, old_expr, new_expr)` that changes an expression at `line`, `ADD(line, new_expr)` that adds a new statement above `line`, and `DEL(line, old_expr)` that allows deletion of an expression in a line. Multiple commands can be joined with the `AND` connector, and finally the bug-revealing test can be executed after modification via the `RUN` command. In addition to experiment commands, the prompt instructs to predict the `<DEBUGGING DONE>` token (`<DONE>` for short in the rest of the paper) if enough information to discern the patch has been gathered, so that we can gauge how confident AutoSD is in its patch. The prompt is detailed enough so that our default LLM, ChatGPT, can follow the instructions zero-shot, i.e., without a concrete demonstration of the full process. On this description of scientific debugging, we add the bug-specific information: concretely, the buggy function/method, the test that reveals the bug, the error message when the bug is executed, and if available a bug report. We add this information as we believe such information would be necessary, if not sufficient, for a human to debug an issue, and thus would likely also help an automated technique to predict appropriate hypotheses and ultimately succeed in debugging.

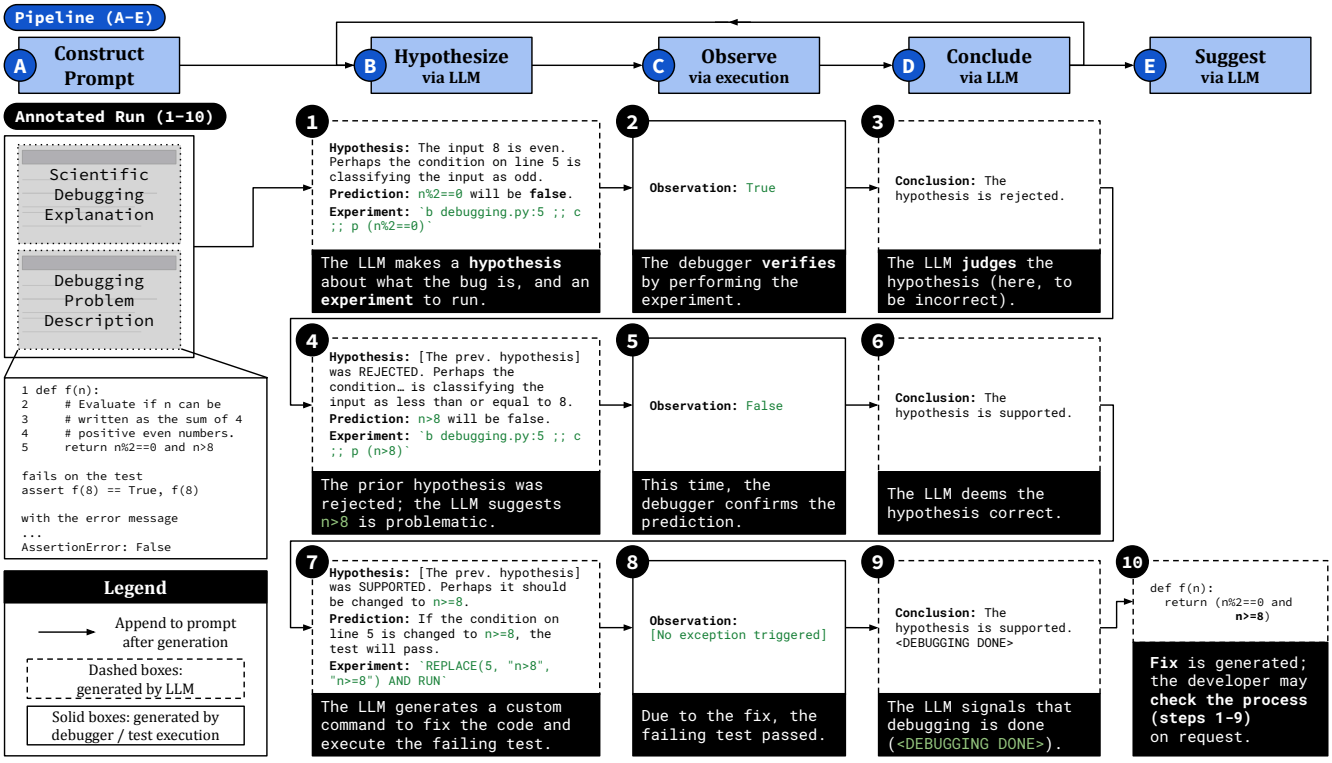


Figure 1: The pipeline and a real example run of AutoSD, with annotations in black boxes and lightly edited for clarity. Given a detailed description of the scientific debugging concept and a description of the bug (A), AutoSD will generate a hypothesis about what the bug is and construct an experiment to verify, using an LLM (B), actually run the experiment using a debugger or code execution (C), and decide whether the hypothesis is correct based on the experiment result using an LLM (D). The hypothesize-observe-conclude loop is repeated until the LLM concludes the debugging or an iteration limit is reached; finally, a fix is generated (E), with an *explanation* (white boxes from (1) to (9)) that the developer may view.

3.2 Hypothesize-Observe-Conclude

With the initial prompt, AutoSD starts iterating over the 'hypothesize-observe-conclude' loop depicted in Figure 1 (B - D). The result of each process is appended to the prompt to allow incremental hypothesis prediction; i.e. when generating the conclusion in 3, the LLM would predict it based on the concatenation of the initial prompt, 1, and 2. We describe each iteration of the loop as a *step*: for example, Figure 1 1 - 3 would make up one step.

Hypothesize. Here, we lead the language model to generate a hypothesis by appending the token `Hypothesis:` to the prompt, so that the language model generates a hypothesis about the bug. We observe that the `Prediction:` and `Experiment:` line headers are also generated in turn by the LLM, due to the detailed description of the scientific debugging process provided by the prompt. The important aspect for the next step is the `Experiment` command, where the language model either generates a debugger command that can be executed by a debugger, or a custom code modification-and-execution script so that the language model can 'test' a certain change. As the document is in Markdown format, the `Experiment` script is wrapped in backticks (`); this script is extracted from the LLM output to get concrete code execution results in the next step.

Examples can be seen in Figure 1 1, 4, and 7 - note that AutoSD also localizes the fault as a part of the hypothesizing process, thus making fault localization explainable as well.

Observe. The generated experiment script is passed to a background process based on traditional software engineering tools that provides real execution results back to the language model, so that we can ground the generation process of AutoSD on real results, and also build credibility for developer presentation. The model can either (i) invoke a composite debugger command by setting a breakpoint and printing a value, or (ii) modify the code and run the failing test with the aforementioned DSL. When executing a debugger command, it is executed via the command-line interface of the language-appropriate debugger, and the output from the last subcommand of the composite command (assumed to be a `print` command) is returned, as in Figure 1 2 and 5. When the breakpoint is within a loop, the debugger collects values at different timesteps of execution and returns them together, e.g. 'At each loop execution, the expression was: [v1, v2, ...]', up to a maximum of 100 values. Meanwhile, upon test execution from a edit-and-execute DSL command, if an exception is raised, the exception type and message are returned as the observation; otherwise, the result '[No exception triggered]' is appended, as in Figure 1 8.

Conclude. Based on the observation, AutoSD invokes the LLM to check whether the hypothesis and the observation are consistent, by having the LLM predict if the hypothesis is rejected (e.g. ❸), supported (e.g. ❻), or undecided due to an unexpected observation. We have the LLM generate the conclusion to maximize flexibility in value interpretation. As described earlier, the LLM may predict a separate <DONE> token at this step if it predicts the debugging process is complete; in such cases, AutoSD would have greater confidence in its output. An example is shown in Figure 1 ❾: on the information that the previously failing test now passes, the LLM concludes that debugging is done. If the <DONE> token is predicted, AutoSD proceeds to generate a fix as in Section 3.3; otherwise the loop restarts with hypothesizing based on the newly available information until a maximum iteration limit s is reached. If <DONE> is not predicted until then, AutoSD is failing to identify the cause of the bug, and we may be more skeptical of the generated patch.

3.3 Fix Suggestion

When AutoSD has completed its interaction with the code, the conclusions to each of the hypotheses are assessed, and rejected hypotheses are automatically removed from the prompt prior to patch generation, as this empirically improved program repair performance in our experiments. Even if rejected hypotheses are not involved when making the fix itself, rejected hypotheses can still be presented to the developer as context for successful hypotheses. We subsequently prompt the LLM to generate a fix using the available information by appending the words "`The repaired code (full method, without comments) is:\n````". This prompt leads the LLM to generate repaired code, based on the information available from the problem description and the code interaction, as in Figure 1 ❿. Identically to other APR techniques, a patch is ultimately generated; what makes AutoSD unique is that it can show its *intermediate reasoning steps* (❶ - ❾) as an *explanation* that can help the developer understand where a patch comes from.

4 EVALUATION SETUP

Here we describe the setup for our empirical evaluation.

4.1 Research Questions

RQ1: Feasibility. While the main focus of our work is to generate a reasoning chain for automated debugging results, good performance in the debugging task itself is also important [17, 28]. We thus seek to answer whether AutoSD achieves performance competitive to prior APR techniques, and when compared to prompting an LLM to immediately predict a fix (this baseline is referred to as LLM-Base in the rest of the paper). We aim to demonstrate that the explainability of AutoSD does not come with a significant performance cost, even as prior reviews on explainable AI describe a tradeoff between interpretability and performance [1]. We evaluate AutoSD on the Almost-Right HumanEval benchmark we construct to mitigate data leakage concerns, and the Defects4J v1.2 and 2.0 benchmarks [15] consisting of real-world bugs.

RQ2: Debugger Ablation. In this research question, we first evaluate whether the performance of AutoSD is better when it indicates that debugging is done via the <DONE> token; as precision is important for practical tools for developers, if AutoSD can indicate when it is likely to be correct, this would aid developer adoption of AutoSD. Based on our confidence-with-<DONE> experiments, we evaluate the performance of AutoSD when debuggers are not used, and observations are 'hallucinated' by the LLM instead of obtained via actual code execution. We evaluate whether under this setting, the <DONE> token continues to be a marker of strong performance.

RQ3: Varying LLM. We evaluate the performance of AutoSD as we vary the LLM that is used. While we empirically found the best performance when using the ChatGPT model, and thus used it as the default setting throughout the rest of the paper, by varying the size of the language model and plotting the performance, we investigate automated repair performance as models improve in terms of parameter size and training sophistication.

RQ4: Developer Benefit. Via our human study, we evaluate whether developers benefit materially from automatically generated explanations by AutoSD, i.e. regardless of their opinion towards explanations. In our human study, participants are given the buggy code, a bug-revealing test, a candidate patch, and half of the time an explanation, and asked to determine whether the patch correctly addresses the issue that the test reveals. We measure the time and accuracy of developers when deciding whether a patch is correct, along with developer answers to the question 'did the explanation help you make the decision?'. We thus hope to evaluate whether developers benefit by being provided explanations.

RQ5: Developer Acceptance. We evaluate whether the explanations of AutoSD are acceptable to developers by asking them six questions on whether they would want to use APR, whether they would want explanations when using APR, and whether AutoSD and each element of its explanation were satisfactory. We thus hope to measure whether developers are willing to use explanations, distinctly from whether their productivity increases from explanations. We additionally perform interviews to identify what developers liked about the explanations of AutoSD, and what could improve.

RQ6: Qualitative Analysis. We provide examples of liked and disliked patch attempts and their corresponding explanations in this research question as further context, along with a breakdown of common failure causes by analyzing a random sample of 25 cases in which all hypotheses generated by AutoSD were classified as incorrect by itself.

4.2 Environment

4.2.1 Evaluating APR Performance. To evaluate the performance of AutoSD, we use four program repair benchmarks. First, the widely-used Defects4J benchmarks [15] version 1.2 and 2.0, which have been used by prior work as a standard benchmark to compare APR techniques [20], are used. We also use the BugsInPy benchmark [35] (abbreviated to BIP in our paper) for the sake of getting real-world Python bugs to evaluate in our human study, but we do not report the performance of AutoSD on BIP as many of its bugs needed additional environment setup not described in the README.

We additionally construct the Almost-Right HumanEval (ARHE) dataset based on the HumanEval Python single-function synthesis benchmark by Chen et al. [3]. We do so in the hopes that it will be free from data contamination concerns, as HumanEval was explicitly made by Chen et al. to avoid data contamination when evaluating their LLM, and was also used to evaluate the recent

GPT-4 model [29]. The ARHE dataset was built by mutating the human solutions in the HumanEval benchmark so that exactly one test fails, making bugs that cause the code to be 'almost' right. We end up with 200 bugs to evaluate with using seven mutators; the detailed composition of the dataset by mutator used is provided in the appendix. For comparison, we additionally compare against a template-based APR baseline that has the reverse mutators of those used to construct the dataset, and randomly applies them to the buggy code. We run this baseline 100 times as it is stochastic. Note that 90 bugs of ARHE are created by deletion or string mutation, and consequently are not reversible by the baseline: all the remaining mutations are reversible and therefore can be fixed by our template-based baseline given sufficient time.

Regarding specific APR parameters, for each dataset we provide AutoSD with the buggy method and generate 10 patches, to match the settings in the large-scale empirical work by Jiang et al. [11], who evaluate the repair performance of multiple large language models and more traditional learning-based APR techniques. We note our setting assumes less exact information and is thus more difficult: Jiang et al. evaluate with perfect statement-level FL, whereas AutoSD uses perfect method-level FL and the bug report. When evaluating the generated patches, we run the tests provided by each dataset for each bug; a fix that makes all tests pass is deemed a *plausible* patch, and plausible patches are manually inspected to see if they are semantically equivalent with the developer patch. Semantically equivalent fixes are deemed *correct*.

AutoSD requires the use of an LLM and a debugger. For the LLMs, we experiment with the CodeGen [27], Codex [3] (code-davinci-002), and ChatGPT (a sibling model to InstructGPT [30]) LLMs, with the ChatGPT LLM being the default model. Different debuggers are used depending on the target language; we use the `jdb` tool for the Java benchmarks (Defects4J v1.2 and v2.0) and the `pdb` tool for the Python benchmarks (ARHE and BugsInPy). The maximum iteration limit, s, is set to 3.

4.2.2 Human Study Parameters. To approximate the real-world impact of AutoSD, we perform a human study by asking participants to review patches, based on the real-world applications of APR [16, 21]. We specifically sampled 12 bugs where AutoSD made a patch that caused the initially failing test to pass: a random sample of six such bugs from the ARHE dataset (which had complete documentation), and six real-world bugs from the BugsInPy Python dataset [35]. In our preliminary studies, we found that reviewing 12 patches could take a long time, so we divided the 12 bugs into two groups of six (each containing three ARHE and three BugsInPy bugs) and randomly assigned participants to solve code review problems from one of the groups. A scheme of the code review screen that was presented to participants is shown in Figure 2; a screenshot of the the survey website can be found in the appendix. Our human study received IRB review exemption (IRB-23-054).

For each code review problem, participants are provided with the buggy code, the bug-revealing (failing) test, along with the patch; they are provided with the explanation in a randomly selected three of the six cases. Each step of the explanation has a header, which is a summary of the hypothesis explaining the bug; the header is color-coded based on the predicted conclusion, with supported/rejected/undecided hypotheses being green/red/yellow, respectively, as in Figure 2. Each header can be clicked to reveal the full reasoning process of AutoSD as depicted in Figure 1. Participants are asked three questions for each patch: (Q1) whether the patch is a correct patch, where they may answer yes, no, or unsure (as a proxy for checking correctness during the code review process [31]); (Q2) a short justification of their decision in Q1, to filter potential bad-faith answers; and (Q3) when an explanation is available, whether the explanation was helpful in making their decision, to measure the differing impact of explanations for different patches.

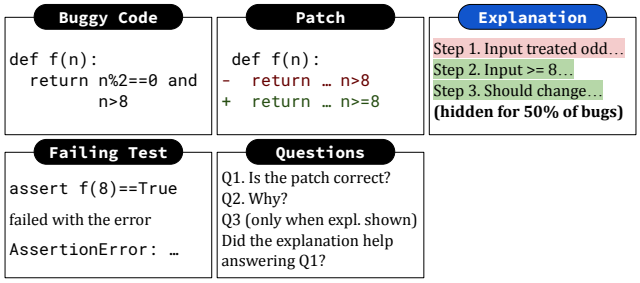


Figure 2: Human Study Screen Scheme

To recruit participants, we advertised the task to both undergraduate and graduate students with at least 1 year of Python experience, as well as professional developers at a company that specializes in software testing techniques. Overall, we recruit 20 participants: eight undergraduate and six graduate students, as well as six professional developers whose career span from 3 to 10 years. Participants start with a briefing of what they should do in the study, solve an example code review problem as practice, and then solve six code review problems in 30-40 minutes in a randomized order. The six code review tasks contain 2 correct and 1 incorrect patches for ARHE and BugsInPy benchmarks, respectively. After conducting a post-questionnaire about their demographics and overall satisfaction with explanations, we perform an interview that lasted about 5 minutes on their impression of the tool for qualitative analysis.

5 EXPERIMENTAL RESULTS

We present the results of empirical evaluation below.

Result	Template-based	LLM-Base	AutoSD
Plausible	85.77 ± 4.20	179	189
Correct	-	177	187

Table 1: Repair results on the ARHE benchmark. The template-based performance is based on 100 reruns, and shows the mean and standard deviation repair performance.

5.1 RQ1: Feasibility

In Table 1, we present the APR performance of AutoSD on the ARHE benchmark when compared with LLM-Base and the template-based baseline. Note that the template-based baseline shows significantly weaker repair performance than both LLM-Base and AutoSD when evaluated under the same conditions; as a result, we

did not assess correctness for the thousands of patches generated, as the upper bound of correctness is the plausible patch count. Additionally, the performance of LLM-Base and AutoSD are similar, demonstrating AutoSD retains the repair performance of the LLM while simultaneously being capable of generating explanations.

Benchmark	Recoder	InCoder	LLM-Base	AutoSD
D4J v1.2	24	41	87	76
D4J v2.0	11	28	110	113

Table 2: Correct repair results on the Defects4J benchmarks. Results for Recoder and InCoder are from Jiang et al. [11].

In Table 2, we present the APR performance of AutoSD on the Defects4J benchmarks when compared against LLM-Base and the best-performing techniques from the empirical study by Jiang et al. [11]: Recoder, a DL-based APR technique [42], and finetuned InCoder [5], a language model from Facebook, which was finetuned with perfect statement-level FL results, and thus uses more exact information than AutoSD. We find that AutoSD again shows competitive performance when compared to other baselines, even those that have more specific information provided. As an additional reference point, when compared against the repair results of Codex on Defects4J presented by Xia et al. [37] we find that AutoSD outperforms Codex using 200 patch candidates (unlike our 10) on both benchmarks under the 'patch function' setting of that paper, which assumes the same FL conditions as our setup.

Answer to RQ1: AutoSD is capable of operating at a competitive level of program repair performance when compared to a diverse set of baselines on three repair benchmarks.

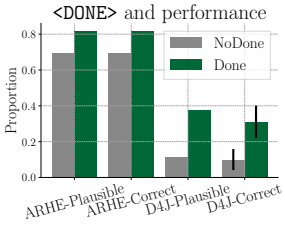


Figure 3: <DONE> & perf.

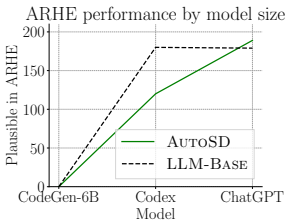


Figure 4: Model Size

5.2 RQ2: Debugger Ablation

This RQ first investigates whether the confidence in a result indicated by the prediction of the <DONE> token actually correlates with better performance. The results are presented in Figure 3. For Defects4J, as it was infeasible to manually label all 1045 plausible patches generated for the dataset, we sampled 100 patches with and without <DONE> to get results. As the figure shows, for both the ARHE and Defects4J datasets, AutoSD shows a higher precision when the <DONE> token is generated as part of a conclusion, indicating that AutoSD can indeed signal when it is likely to generate a plausible or correct patch. Furthermore, for bugs where a plausible patch was generated and the <DONE> token was predicted, 89% were correctly fixed, while for bugs with plausible patches but without <DONE> predictions 82% were correctly fixed.

We also investigate the performance when the debugger/code execution results are also predicted by the LLM, instead of being obtained via concrete execution, for the ARHE dataset; would the <DONE> token still predict good performance? In this 'debugger hallucination' scenario, <DONE>-predicted solutions were actually 11%p *less* likely to be plausible; this is in contrast to using actual code execution results, where <DONE>-predicted solutions are 12.4%p more likely to be plausible. Furthermore, individual runs became much less likely to be plausible: while 73% of the individual AutoSD runs would yield a plausible patch, only 63% would when the debugger was ablated. Thus, incorporating code execution contributes to the reliability of AutoSD; we later demonstrate in RQ5 that developers found real code execution results useful as well.

Answer to RQ2: AutoSD can indicate when its answers are more likely to be correct with the <DONE> token, which we also use to verify the utility of debugger use.

5.3 RQ3: Varying LLM

In Figure 4, we depict the performance of AutoSD as different underlying LLMs are used, with the x axis showing different LLMs roughly sorted in terms of number of parameters and the technical advancement of training, and the y axis showing the performance of AutoSD when using the LLM on the ARHE benchmark. The performance of AutoSD is depicted along with the performance of simply querying the LLM to fix the bug. As shown, the performance of AutoSD rapidly improves and ultimately becomes comparable to the performance of LLM-Base, suggesting that AutoSD shows better performance when using stronger language models; for smaller models such as CodeGen-6B, repair itself fails in a zero-shot setting, as in our experiments it would simply return the original buggy code. (We confirm that the model implementation works by also evaluating in a few-shot setting for CodeGen-6B; it could fix 44 bugs in that case.) Thus, we may speculate that as language models improve, the performance of AutoSD will also become stronger.

Answer to RQ3: As the underlying language model improves, the performance of AutoSD also increases.

5.4 RQ4: Developer Benefit

In this section, we evaluate whether developers benefit from explanations in a way that is unlikely to be swayed by a participant's opinion about explanations. The results of measuring the code review time, accuracy, and whether the explanation was rated as helpful in making the decision are presented in Figure 5.

First, looking at the amount of time that it took to solve the code review problems, we find that the time it took to solve a problem was generally similar between the case where there was no explanation and when there was an explanation. There is no case where the difference is statistically significant, despite the explanations of AutoSD providing more information than the case without explanations, and thus potentially requiring more processing time from developers.

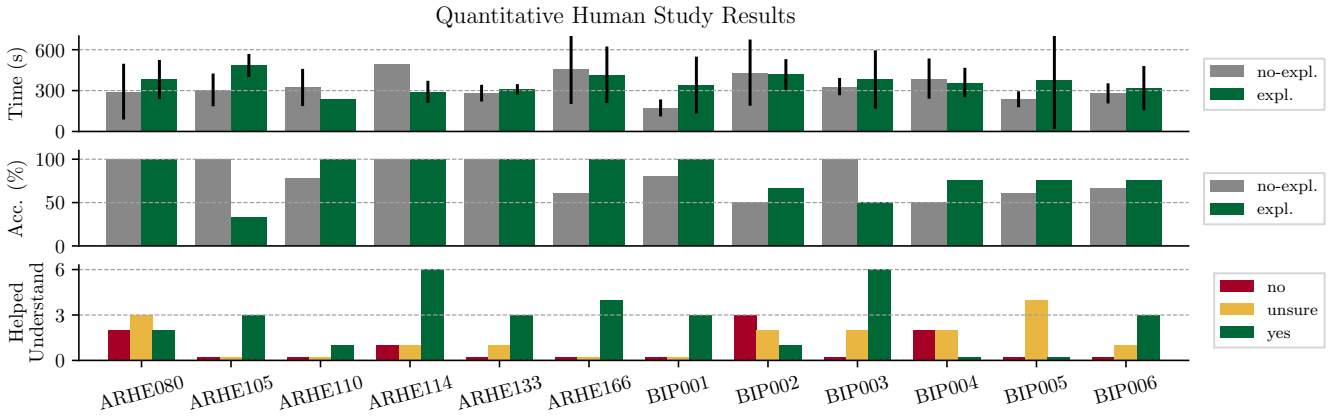


Figure 5: Developer performance on code review tasks with and without explanations from AutoSD, and explanation ratings

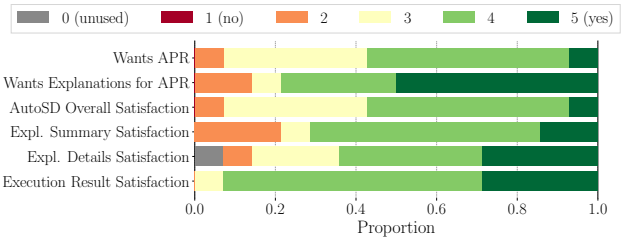
Regarding the accuracy with and without explanations, participants were more accurate when solving the same problems with explanations than without explanations in seven cases, with five of them being concentrated in the real-world BugsInPy benchmark. These results demonstrate that AutoSD could have a positive impact on real-world developer productivity when using APR, as the judgment quality improved when evaluating real-world bugs while requiring roughly the same amount of developer time. Meanwhile, there are two cases where the use of explanations lead to a drop in accuracy: ARHE105 and BIP003. For BIP003, we found that the respondents became more cautious after looking at the explanation, and answered that they needed more information to judge it. Meanwhile, for ARHE105 the participants who answered incorrectly accepted the reasoning of AutoSD without significant scrutiny. While this was a somewhat rare incidence that happened in one of the 12 randomly sampled problems, it highlights the need of further research to identify potentially misleading reasoning. Additionally, developer accuracy improved with explanations on the two incorrect patches from BIP (BIP002 and BIP004) meaning developers are not blindly accepting patches with explanations.

On whether the participants found the explanations helpful in their decision-making, in eight of the twelve questions developers noted that the explanations were actually helpful when coming to their conclusion, underscoring the psychological benefit that providing explanations for patches holds.

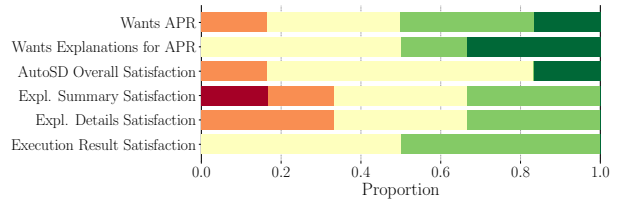
Answer to RQ4: When exposed to explanations generated by AutoSD, human participants could process patches in roughly the same time, while achieving a higher accuracy in five of the six of the real-world bugs. They also rate the explanations as helpful in two-thirds of all bugs.

5.5 RQ5: Developer Acceptance

The results of our post-questionnaire are presented in Figure 6. To our surprise, there was a discrepancy in satisfaction of AutoSD between students and professional developers: while more than half of the students were satisfied with AutoSD, only one of the six developers were satisfied. We use these differing results as an



(a) Satisfaction among students.



(b) Satisfaction among professional developers.

Figure 6: Human study post-questionnaire results by group.

opportunity to discuss the strengths and potential improvements of AutoSD-generated explanations.

What did students find appealing about the explanations of AutoSD? Ten out of the 14 student participants noted that they 'missed' the explanations when they were not available. When asked why they wanted to see the explanations in these cases, and how they used explanations when they were available, students described a wide range of thought processes that were aided by the existence of explanations. One common pattern was to think through the patch by oneself, then comparing one's internal thoughts to the provided explanation; one participant referred to the explanation as useful because it could function as a 'rubber duck'.[2] Another common usage of explanations was to look at the explanation to discern where to focus effort on, and thus guide the direction of judgment. Other students would use the explanation to gain a better

[2]See https://en.wikipedia.org/wiki/Rubber_duck_debugging.

understanding of what the code was intended to do. We thus argue that a strength of AutoSD-generated explanations is that **they can accommodate a diverse set of thought processes**, potentially aiding a wide range of developers.

Meanwhile, another usage pattern was to look at the experiments and observations within the explanations to get a concrete idea of what the values are at certain points, and use those values to build a mental model of how the bug happened. This points to another strength of AutoSD, which is that it **incorporates actual values in its explanations**: in Figure 6 (a), we note that more than 90% of students thought that the addition of execution results improved their trust in the explanations.

On the other hand, professional developers showed a more mixed attitude towards the explanations of AutoSD. It is noteworthy that developers are not opposed to explanations themselves: half agreed or strongly agreed that explanations would be important when using an APR tool (Figure 6 (b)), highlighting the importance of the problem. When asked why they found the explanations of AutoSD left more to be desired, one suggestion was that the current explanations would be more useful if they were connected with "business logic" or specifications, a suggestion echoed by one of the student participants as well. The professional developers argued that without such connections, the explanations needed to be verified rigorously and even after that were of limited value. Thus one potential direction of improvement would be to **integrate explanations with existing development artifacts like specification documents**.

Another common suggestion was to improve the interface of the tool: developers noted that they might use the tool if it was attached to an IDE, and that the explanations were too wordy. This feedback suggests that to improve developer satisfaction, we may consider **integrating explanations to platforms that developers frequent** (as also suggested by Kochhar et al. [17]), and further study the specifics of explanations that developers find satisfactory.

Looking at the overall statistics, we find that 70% of participants agreed that explanations were an important factor when using program repair, and 55% found the scientific debugging details (Expl. Details Satisfaction of Figure 6) satisfactory, showing that a majority of participants agreed with the overall motivation and formulation of AutoSD.

Answer to RQ5: While the explanations of AutoSD are capable of accommodating diverse thought processes and improving developer trust by using concrete execution results, they could be further improved by enhancing the interface and by linking to specifications.

5.6 RQ6: Qualitative Analysis

What do the explanations generated by AutoSD look like? In addition to the example embedded in Figure 1, we provide two reasoning traces generated by AutoSD that were liked (BIP006 - 75% liked) and disliked (BIP002 - 16% liked) in the human study from the real-world BugsInPy problems. On the left of Figure 7, we show a liked explanation, along with a condensed failing test and the generated fix. Looking at the patch, the developer will see that a `.lower()` call was added; without an explanation, this fix can appear spurious. In contrast, by providing a rationale on why AutoSD focused on this area, participants could swiftly identify whether this fix was related to the test. For example, Student-6 said "I first looked at the explanation, which helped me identify which part of the code to look at". The subsequent experiment confirms that an uppercase `'Chunked'` header within the program state, which is the source of the bug. These execution results helped participants understand the bugs, e.g. Student-11 who noted that "expression values were useful in making decisions". Overall, this patch was correct, and the explanation aided developer comprehension and built trust. While we provide a simple example from the human study, we also note that AutoSD works on more complex bugs as demonstrated in Section 5.1, and provide additional examples in the appendix.

Attempts at hypothesizing can fail as well. The right side of Figure 7 depicts an case where AutoSD fails to validate any hypotheses. While AutoSD initially generates a hypothesis about appending in the wrong order, the line that is suggested in the experiment is actually not covered; as a result, the debugger provides feedback that the breakpoint was not covered. This is one of the most common failure causes - our analysis on 25 cases where all hypotheses were rejected revealed that in 13 of the 25 cases, breakpoints suggested by AutoSD were never hit, and consequently AutoSD could not get results for generated experiments. In BIP002's case, instead of looking for new breakpoints that could be covered by the test, the LLM starts suggesting that the test is wrong. Ultimately, while a fix is generated, the explanation has little connection to the patch, and as a result the human study participants rated the explanation as unhelpful; the patch itself is plausible but incorrect as well. Nonetheless, the example also illustrates how bad explanations can still lead to better decision-making: developers may see that the foundations of the patch are weak, and be (rightly) more suspicious about the patch. In this context, it is noteworthy that developers who saw the explanation of BIP002 more accurately assessed it (Figure 5). Other failure modes include generating an invalid experiment expression (2/25) or adding multiple print commands in the experiment script when the infrastructure of AutoSD only allows one print command, causing inaccurate hypothesis rejection (2/25).

Answer to RQ6: AutoSD can generate helpful explanations on its patches, but the reasoning process may fail as well. A common failure cause is an inability to identify the right breakpoints.

6 DISCUSSION

This section provides threats and limitations of our work.

6.1 Threats to Validity

Internal Validity concerns whether the analysis supports claims about cause and effect. Potential threats include incorrect implementations, inaccurate patch correctness assessment, and the risk of biased responses in our human study. To mitigate the impact of the first two concerns, we plan to make our implementation and repair results publicly available for scrutiny. For our human study, in addition to gathering developer sentiment about the generated explanations (which included occasional negative feedback), we also find that participant accuracy improved in five of the six BugsInPy problems, which is a result difficult to be due to bias.

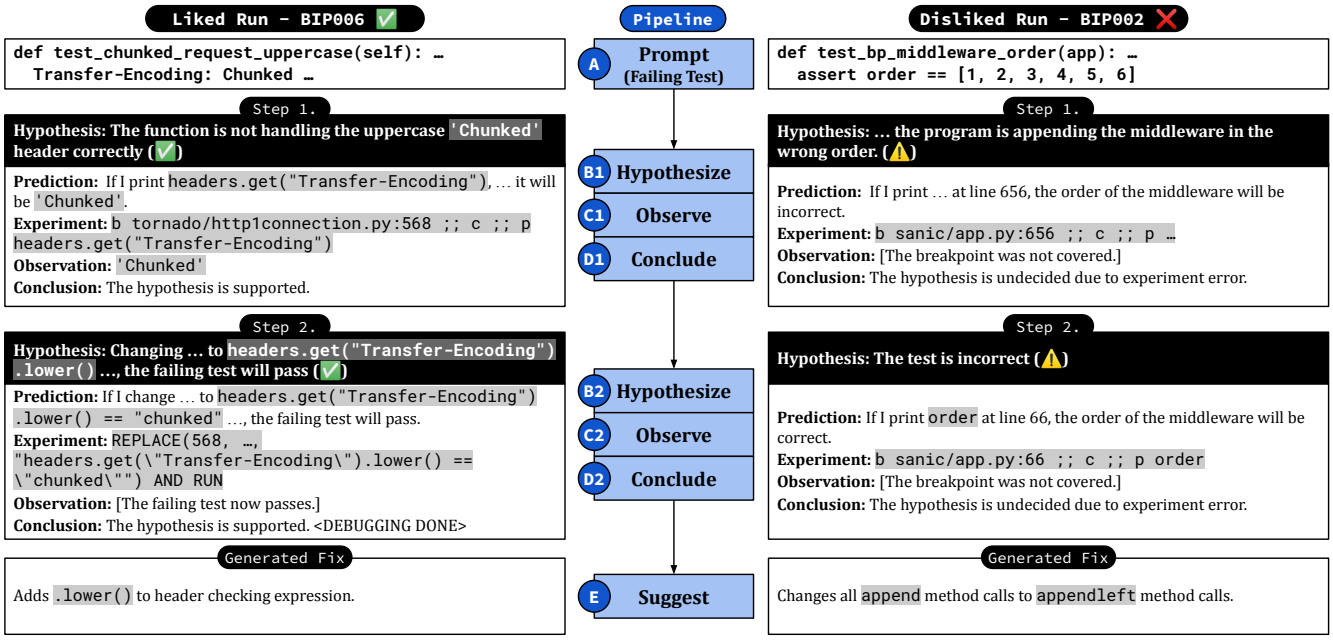


Figure 7: Example successful and unsuccessful repairs and explanations of AutoSD from the human study.

External Validity concerns whether the results presented in this paper may generalize to other results. A particular concern when using large language models is that their training data may include segments of the evaluation data. To mitigate this issue, we newly constructed the ARHE dataset for repair and evaluated AutoSD on that benchmark. Furthermore, our explanations were likely never within the training data, as developers usually describe code with less of a structure than Scientific Debugging prescribes, even if they think along the lines of it.

6.2 Limitations

AutoSD has a number of limitations that we would like to highlight. First, to enable multi-step interaction with code, both the language model and debugger must be invoked multiple times, which increases the repair time of the technique; in our experiments, AutoSD could take about five times longer to generate a patch when compared to LLM-Base. Nonetheless, given the significant developer demand for explanations of automatically generated patches as shown in Figure 6, we believe that the additional cost needed to build explanations for patches is justified. Second, as a step towards explainable automatic debugging, we evaluated in the setting where method-level FL was done, and AutoSD would then perform statement-level FL in an explainable manner. Our main focus in this paper was to establish that AutoSD can generate explanations that aid developers in practice; we hope to work on explainable method-level FL in future work. On a related note, our technique can only handle single-method bugs as of now; incorporating a wider range of information to handle more complex bugs is also an interesting research direction. Finally, the generated explanation may occasionally lend credibility to incorrect patches; by allowing our technique to indicate its confidence in its output and demonstrating that confidence is correlated with correctness, we take the first steps to address this issue. Furthermore, our explanation includes concrete code execution results, aiding developer decision-making (Figure 6).

7 CONCLUSION

In this paper, we summarize the importance of explanations for automated debugging results as revealed by prior studies, and the lack of automated techniques capable of providing adequate explanations for humans. We argue this is due to a lack of automated debugging techniques that deduce in a human way, and bridge this gap between automatic and manual debugging practices by using LLMs to emulate the Scientific Debugging process. We demonstrate that AutoSD is capable of achieving competitive repair performance when compared to other repair baselines, while having favorable properties for practical use such as an indication of confidence in the output. The repair performance of AutoSD also improves as language models become more capable, suggesting the performance and availability of explanations may improve as language models get better. Finally, our human study reveals that the automatically generated explanations could improve developer assessment of patches, with a majority of students also expressing that they 'missed' the explanations when they were not available. The interviews we performed show that the explanations AutoSD generates could aid a wide range of developer thought patterns, and that they could be improved via tighter integration into the development process, such as making connections to written specification. Overall, we believe that the rapid improvement in language model capabilities can be harnessed to significantly ease developer use of automated techniques, and we hope to develop more human-friendly automated debugging techniques as future work.

REFERENCES

[1] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. 2020. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information fusion* 58 (2020), 82–115.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[4] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2018. Explainable Software Analytics. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results* (Gothenburg, Sweden) *(ICSE-NIER '18)*. Association for Computing Machinery, New York, NY, USA, 53–56.

[5] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[6] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. PAL: Program-aided Language Models. *arXiv preprint arXiv:2211.10435* (2022).

[7] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[8] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65.

[9] John D. Gould. 1975. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies* 7, 2 (1975), 151–182. https://doi.org/10.1016/S0020-7373(75)80005-8

[10] J. Jiang, Yingfei Xiong, H. Zhang, Q. Gao, and X. Chen. 2018. Shaping program repair space with existing patches and similar code. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018).

[11] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. arXiv:2302.05020 [cs.SE]

[12] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. arXiv:2302.01857 [cs.SE]

[13] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 135–146. https://doi.org/10.1109/ASE.2017.8115626

[14] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 467–477.

[15] 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* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[16] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, Tracy Hall, Saemundur Haraldsson, and John Woodward. 2021. On The Introduction of Automatic Program Repair in Bloomberg. *IEEE Software* 38, 4 (2021), 43–51. https://doi.org/10.1109/MS.2021.3071086

[17] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/2931037.2931051

[18] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180.

[19] Brian Y. Lim, Anind K. Dey, and Daniel Avrahami. 2009. Why and Why Not Explanations Improve the Intelligibility of Context-Aware Intelligent Systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 2119–2128.

[20] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 615–627. https://doi.org/10.1145/3377811.3380338

[21] 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)*. 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[22] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezze. 2011. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Transactions on Software Engineering* 37, 4 (2011), 486–508. https://doi.org/10.1109/TSE.2010.93

[23] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701. https://doi.org/10.1145/2884781.2884807

[24] M. Monperrus. 2019. Explainable Software Bot Contributions: Case Study of Automated Bug Fixes. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 12–15. https://doi.org/10.1109/BotSE.2019.00010

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

[26] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 153–162. https://doi.org/10.1109/ICST.2014.28

[27] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint* (2022).

[28] 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2228–2240. https://doi.org/10.1145/3510003.3510040

[29] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[30] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155* (2022).

[31] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/3183519.3183525

[32] Benjamin Siegmund, Michael Perscheid, Marcel Taeumel, and Robert Hirschfeld. 2014. Studying the Advancement in Debugging Practice of Professional Software Developers. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 269–274. https://doi.org/10.1109/ISSREW.2014.36

[33] Chengnian Sun and Siau-Cheng Khoo. 2013. Mining Succinct Predicated Bug Signatures. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 576–586.

[34] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv* abs/2201.11903 (2022).

[35] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560.

[36] Emily Rowan Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Haraldsson, John Woodward, Serkan Kirbas, Etienne Windels, Olayori McBello, Abdurahman Atakishiyev, Kevin Kells, and Matthew Pagano. 2022. Towards Developer-Centered Automatic Program Repair: Findings from Bloomberg. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1578–1588. https://doi.org/10.1145/3540250.3558953

[37] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical Program Repair in the Era of Large Pre-trained Language Models. *arXiv preprint arXiv:2210.14179* (2022).

[38] 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 European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971.

[39] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 416–426.

[40] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).

[41] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.

[42] 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*. Association for Computing Machinery, New York, NY, USA, 341–353.

Appendix for Explainable Automated Debugging via Large Language Model-driven Scientific Debugging

1 DISCUSSION OF EXPLAINABLE FAULT LOCALIZATION

In this section, we make the argument that fault localization needs explanations as well, and that while there are a number of fault localization techniques that argue that they are explainable and can be helpful to developers, there are few human studies. To start, as mentioned in the paper, Kochhar et al. [7] survey that 85% of developers want explanations for fault localization. If anything, the need for explanations in fault localization is greater, as while for automated program repair a suggested patch is actionable (one may accept it, reject it, or inspect it) it can be unclear what to do with a fault localization result, similarly to what has been noted for defect prediction results [9]. It is noteworthy that in Kochhar et al.'s survey, developers also relate the need of an explanation to fixing and 'actionability': one developer notes that "...to make a decisions about bug fixing I want to *exactly* know why the automated tool thinks that the code have a bug [sic]", for instance.

Some commonly used fault localization techniques include Spectrum-Based Fault Localization (SBFL) and Information Retrieval-based Fault Localization (IRFL). SBFL analyzes the coverage patterns of failing and passing tests, and uses various formulae to suggest the statements that are most correlated with the fault [5]. Meanwhile, IRFL uses bug reports or failing tests and analyzes the textual similarity between those artifacts and the source code to identify the file or method most correlated with the failure description [8]. While the reasoning traces of these techniques can be presented to developers, Kochhar et al. note that "these basic rationales are not likely to be sufficient to help practitioners", while citing Parnin and Orso [13], who questioned the utility of fault localization techniques via a human study. Recent improvements in fault localization techniques include Mutation-Based Fault Localization (MBFL) [12], which mutates statements and observes the changes in test behavior to identify likely fault locations, and fault localization based on machine learning [10], which uses features from an assortment of FL techniques and uses machine learning to predict which locations are likely to be faulty. Similarly to our observations about automated program repair, none of these techniques deduce in a humanlike way, and as a result it is difficult to expect that presenting the reasoning trace of any of these techniques would help understanding the results of the technique (for machine learning-based fault localization, it is also unclear if a reasoning trace exists in the first place).

At the same time, our understanding is that there is still more explainable fault localization research than in program repair. We look to three surveys [1, 14, 16] to identify relevant research. `explain` was proposed by Groce [4], which compares a failing test to the maximally similar passing test to isolate where program values diverge. Early work of Zeller was also in a related direction, in which delta debugging was applied on internal program states to perform fault localization [2, 17]. Cleve and Zeller [2], for example, note in the paper's conclusion that developers would "not only know that a test has failed, but also *why* and *where* it failed", indicating their interest in explaining bugs and fault localization results as well. More recently, Sumner and Zhang [15] use slicing to make the state replacement technique of Zeller more precise and thus more accurately explain differences. While we are inspired by this line of work, the aforementioned literature did not perform user studies on the provided explanations that we may compare the effects against. Furthermore, relative to AutoSD there are significant discrepancies on how debugging is done: in AutoSD, the LLM automatically generates hypotheses about what is wrong with the code and extracts values accordingly, whereas in aforementioned work the values of all variables are inspected to isolate the bug-causing change. Whyline [6] allows

developers to ask 'why' and 'why not' questions to a debugging system and get answers; unlike AutoSD, the focus is not on automated debugging, and human developers are still making the hypotheses. As a post-hoc technique that explains why a location might be buggy (but being incapable of actually describing *why* a tool located that particular location) Mahbub et al. propose Bugsplainer [11], which uses Neural Machine Translation (NMT) to train a Transformer model that translates an identified location to a likely commit message.

Overall, it seems that the observation that Alipour [1] made more than ten years ago that "the most suitable level of abstraction for explaining failures is unknown" appears to still be the case; we hope our manuscript can provide some hints to the answer.

2 ARHE BENCHMARK MUTATOR BREAKDOWN

The breakdown of the ARHE benchmark by mutation used to generate each bug is presented in Table 1. We also describe the details of each mutator here, and compare them to mutators in PIT [3], a widely used mutation testing tool. 'Integer Literal Changer' will change literal `0` constants to `1` constants, and vice versa, which shows similar behavior to the 'Inline Constant Mutator' of PIT. 'If Remover' will remove the then-block or else-block of an if statement; if it has no remaining children, the if statement itself will be removed, similarly to 'Remove Conditionals Mutator' of PIT. 'String Literal Changer' will make a string literal empty, lower-case, or upper-case; making the string literal an empty string is not reversible, but whether the lower-casing or upper-casing can be applied in the reverse to get the original code differs from problem to problem. The generation of empty strings is similar to the 'Empty returns Mutator' of PIT. 'Operator Changer' will change pluses to minuses, along with similar operations, similarly to the 'Math Mutator' of PIT. 'Binary Operator Remover' will remove a binary operator and only leave one of the operands, similarly to the 'Arithmetic Operator Deletion Mutator' of PIT. 'Augmented Assignment Changer' will change `+=` to `-=`, vice versa, etc., similarly to the 'Increments Mutator' of PIT. 'If negator' will add a `not` to an `if` condition, similarly to the 'Negate Conditionals Mutator' of PIT.

In Table 1, the 24 bugs from If Remover and 24 bugs from Binary Operator Remover are not reversible; furthermore, we manually determine that 42 of the 63 String Literal Changer bugs are not reversible, making for a total of 90 bugs that cannot be repaired by applying the same mutation set.

Mutator	Number
Integer Literal Changer ○	45
If Remover □	24
String Literal Changer Δ	63
Operator Changer ○	40
Binary Operator Remover □	24
Augmented Assignment Changer ○	3
If Negator ○	1

Table 1. ARHE benchmark breakdown. Reversible mutators are marked with ○, unreversible mutators are marked with □, and occasionally reversible mutators are marked with Δ.

3 SCREENSHOT OF WEBSITE

A screenshot of the human study screen is provided in Figure 1. Note that the original buggy code and error message are shown on the left column, the patch is suggested in the middle, the explanation is shown on the right, and the questions are presented on the bottom center of the webpage, similarly to Figure 2 of our manuscript.

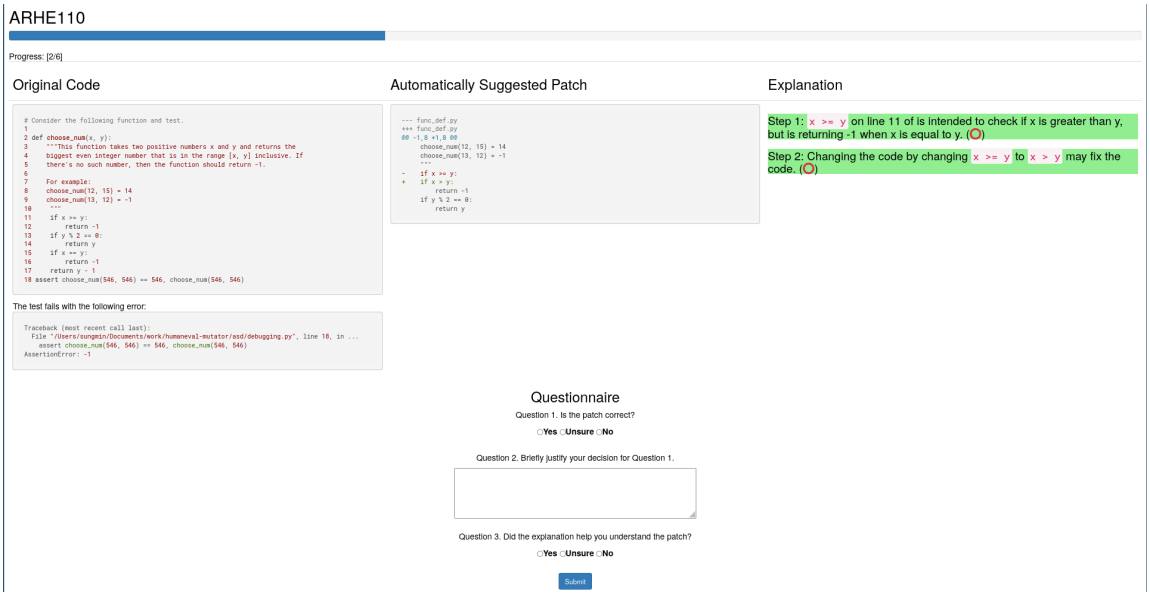


Fig. 1. Screenshot of the human study webpage.

4 SCIENTIFIC DEBUGGING PROMPT

The box below shows the the Scientific Debugging description prompt used for the Defects4J benchmark.

I am going to use the scientific method to debug the problem below (as written by Zeller, 2009) by describing the hypothesis/prediction/experiment/observation/conclusion. This can be done by:
Hypothesis: An explanation for the buggy behavior. Hypotheses are the key aspect of the approach, and should be detailed and written with care. Hypotheses should build upon all previous information; repeating previous hypotheses is thus strongly discouraged. Some examples are provided below.
- Example hypothesis 1: "Given that [information], the method is [overall erroneous behavior]. Specifically, I think it is because ‘c>b‘ on line 4321 of method ‘foo‘ is intended to [desired behavior], but is [erroneous behavior]."
- Example hypothesis 2: "As the previous hypothesis was rejected, we now know ‘c>b‘ on line 4321 of the method ‘foo‘ is likely not the culprit. Looking elsewhere, perhaps ‘x.append(y)‘ should do [desired behavior], but is doing [erroneous behavior]."
- Example hypothesis 3: "Because the previous hypothesis was supported, I think changing the code by changing ‘c>b‘ to ‘c>b && a <= d‘ may fix the code."
- Example hypothesis 4: "It seems the previous experiment ended in an error, we may need to try a different experiment. Perhaps the experiment can be refined by [new experiment]."
Prediction: A specific value or symptom that would be observed if the hypothesis is correct. Depending on the hypothesis, one may make the prediction that a test will pass. Make specific considerations for expressions within loops.

- Example prediction 1: If I use the debugger to print [expr], while given the input and its intended role indicates that its value should be [desired value], it will not be so; that is, when I stop the debugger at line lineno, ‘[verifying_expr]‘ will be true.
- Example prediction 2: If I change [expr] to [new_expr], the test will pass.
- Example prediction 3: If I change the code to utilize the new variable, the test will pass.

Experiment: A specific ‘jdb‘ script that would check whether the prediction made is true. Make sure the line points to an actual statement (not a bracket).
- Example 1: (pdb script): ‘stop at org.not.a.test.class.file:lineno ; run ; print [verifying_expr]‘
- Example 2: (edit script, REPLACE/ADD/DEL available): ‘REPLACE(4321, "c>b", "c>b && a <= d") AND ADD(4323, "a+=1;") AND RUN‘

Observation: The output of the ‘jdb‘ script. Example: ‘[value]‘

Conclusion: A judgement whether the hypothesis is true based on the observation. Also add <DEBUGGING DONE> when the hypothesis confirmed leads to a concrete program fix.
- Example: The hypothesis is (supported/rejected/undecided due to experiment error). [When a test passed, add <DEBUGGING DONE>.]

5 DEFECTS4J AUTOSD EXAMPLES

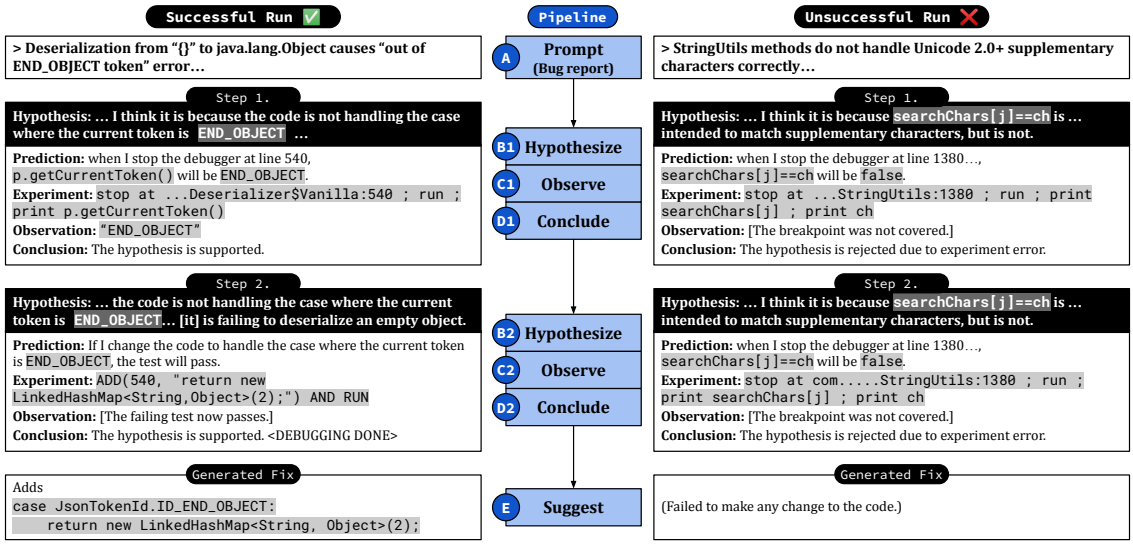


Fig. 2. Example AutoSD runs from Defects4J bugs.

A reasoning trace that ultimately lead to a correct patch and one that did not are presented in Figure 2. In the left case, AutoSD hypothesizes that the bug is happening when the current token is `END_OBJECT`, and generates an experiment to confirm that this is the case. As this is actually the case, it proceeds to search for what behavior would lead to the failing test to pass in Attempt 2. Combining these two steps together, it generates a patch identical (in this method) to the developer patch, and that makes all tests in the test suite pass. Meanwhile, on the right, another example of failing

to identify the right breakpoint is provided. In this case, the same hypothesis and experiments are parroted, leading to no improvement.

REFERENCES

[1] Mohammad Amin Alipour. 2012. Automated fault localization techniques: a survey. *Oregon State University* 54, 3 (2012).

[2] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 342–351.

[3] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452.

[4] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. 2006. Error Explanation with Distance Metrics. 8, 3 (jun 2006), 229–247.

[5] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 467–477.

[6] Amy J. Ko and Brad A. Myers. 2008. Source-Level Debugging with the Whyline. In *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering* (Leipzig, Germany) *(CHASE '08)*. Association for Computing Machinery, New York, NY, USA, 69–72.

[7] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/2931037.2931051

[8] 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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 314–325.

[9] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. 2013. Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 372–381.

[10] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180.

[11] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 2023. Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation. arXiv:2212.04584 [cs.SE]

[12] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 153–162. https://doi.org/10.1109/ICST.2014.28

[13] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 199–209.

[14] Alexandre Perez, Rui Abreu, and Eric Wong. 2014. A survey on fault localization techniques. (2014).

[15] William N Sumner and Xiangyu Zhang. 2013. Comparative causality: Explaining the differences between executions. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 272–281.

[16] 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. https://doi.org/10.1109/TSE.2016.2521368

[17] Andreas Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Charleston, South Carolina, USA) *(SIGSOFT '02/FSE-10)*. Association for Computing Machinery, New York, NY, USA, 1–10.