Does Back-stepping Help Programmers Debug?

LÉONARD OEST O'LEARY, University of Montreal, Canada
JINGYUE ZHANG, University of Montreal, Canada
OLIVIER MELANÇON, University of Montreal, Canada
IAN ARAWJO, University of Montreal, Canada
MARC FEELEY, University of Montreal, Canada

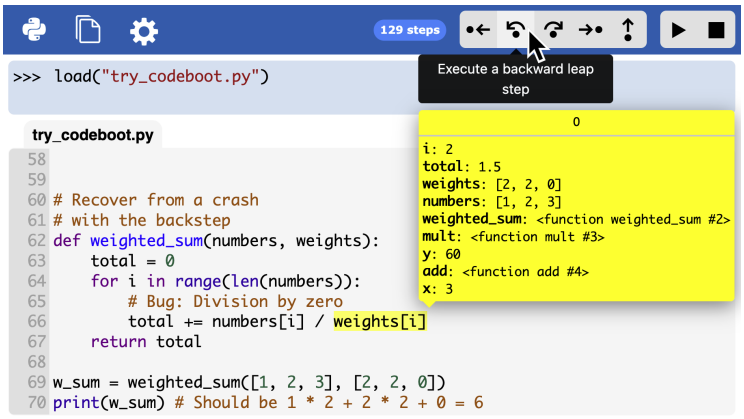

Fig. 1. Showcase of the reverse debugging feature in codeBoot.

Reverse debugging, also called time travel debugging, enables programmers to navigate forward and backward during the execution of a program. Reverse debuggers typically introduce this feature through buttons allowing the programmer to *back-step* to a previous execution point. Although this feature is used in multiple user interfaces, to the best of our knowledge, only anecdotal evidence exists to suggest that back-stepping helps programmers to debug.

To better understand the potential effects of back-stepping, we ran a within-subjects study with 18 participants with Python knowledge who regularly use debuggers. Three conditions were compared through ablations of the codeBoot Python environment: (1) no debuggers, (2) only forward debugging, and (3) forward and backward debugging. Qualitative results show that participants found back-stepping transformative, even calling it a "cheat code", while quantitative results also provide evidence of usefulness, albeit to a lesser degree. Usefulness of the back-stepping button as an interface to reverse debugging is also discussed.

CCS Concepts: • **Human-centered computing** → **User interface programming**; • **Software and its engineering** → **Software testing and debugging**; *Runtime environments.*

Additional Key Words and Phrases: Time travel debugging, Reverse Debugging, User interface programming

Authors' Contact Information: Léonard Oest O'Leary, leonard@oestoleary.com, University of Montreal, Montréal, Québec, Canada; Jingyue Zhang, jingyue.zhang@umontreal.ca, University of Montreal, Montréal, Québec, Canada; Olivier Melançon, olivier.melancon.1@umontreal.ca, University of Montreal, Montréal, Québec, Canada; Ian Arawjo, ian.arawjo@umontreal.ca, University of Montreal, Montréal, Québec, Canada; Marc Feeley, feeley@iro.umontreal.ca, University of Montreal, Montréal, Québec, Canada.

2025. ACM XXXX-XXXX/2025/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Léonard Oest O'Leary, Jingyue Zhang, Olivier Melançon, Ian Arawjo, and Marc Feeley. 2025. Does Back-stepping Help Programmers Debug?. 1, 1 (December 2025), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Most bugs arise from mismatches between a programmer's understanding of a program and its actual behavior during execution. Debuggers allow observation of the program's inner workings to improve understanding of issues and correct misunderstandings in the programmer's mental model. More specifically, debuggers allow programmers to control the pace of program's execution, stopping to inspect variables and function call hierarchies. This navigation through the evolving state of the program is typically enabled by two interfaces: *breakpoints*, which allow execution to run until selected points, skipping over multiple lines for a fast navigation, and *stepping* for a slower, careful navigation, going through each line or expression of the program.

Reverse debugging can be seen as a natural extension of classical forward debugging, allowing programmers to freely navigate between the past and future of a program's execution, sometimes referred to as time travel debugging. Conceptually, it refers to the underlying capability to revisit past execution states. User interfaces typically expose this capability through *back-stepping*, which can be invoked through different interaction mechanisms such as buttons in visual debugger or commands in a command-line debugger. *Back-stepping* is generally assumed to be beneficial, but only anecdotal evidence supports this claim.

To the best of our knowledge, this paper presents the first study on back-stepping in a controlled experiment. Throughout the study, we sought to understand the impact of back-stepping *as a feature*, trying to answer questions such as: *Do programmers like back-stepping? Are programmers more productive while using back-stepping?* and *How do programmers use back-stepping?* Understanding back-stepping goes beyond validating the usefulness of the tool, and can give insights on how to design useful interfaces for reverse debugging.

The contributions of the study can be summarized as follows:

(1) We present the first controlled mixed-methods study examining the effects of back-stepping in debugging, providing empirical evidence that reverse debugging can be beneficial for experienced programmers under certain task conditions.
(2) We characterize how access to back-stepping changes debugging behavior, showing different investigation approaches compared to forward-only debugging, while also revealing contextual dependencies.
(3) We identify limitations of the *back-stepping* button as a primary interface to reverse debugging, and derive design implications that motivate alternative approaches, such as *undo-style* navigation of execution history.

2 Related Work

Reverse debugging is not a novel idea, many debuggers natively support back-stepping. Command-line debuggers with reverse debugging include `GDB` [2], Haskell's interpreter [5, 6], OCaml's debugger [8, 15], `RR` [14]. Many enterprise solutions also enable reverse debugging: Visual Studio (IntelliTrace [12] and Microsoft Time Travel Debugger [13]), undoUDB [19], revdebug [16]. Even some educational programming environments include reverse debugging: JIVE [9], PyTutor [3]. Back-stepping managed to slip into the most used debuggers and learning tools in the world, but few developers are aware of it [17].

Although back-stepping has a wide adoption among debuggers, user studies are sparse. This gap in empirical evidence limits our understanding on how back-stepping is employed by users and what are the possible limitations of this technology.

Educational interfaces. Thayer, Guo and Reinecke [18] published a large study to evaluate the impact of culture on debugging practices using the PyTutor [3] website. They found an inverse correlation between debugging success and the usage of the back-stepping button. This raises doubts about whether back-stepping is helpful for programmers. However, the authors pointed out the possible limitation that programmers who used back-stepping might have been ones who struggled with debugging with the other tools at hand [18, p. 7]. Our work is a continuation of Thayer, Guo and Reinecke's, evaluating the impact of back-stepping in a more controlled environment (comparing against a control group), with more challenging tasks, and experienced programmers.

Multiple educational interfaces use combinations of creative and novel programming interfaces to bring programming to a broad audience, sometimes using reverse debugging to provide novice with the ability to step forward and backward. However, user studies in these works often evaluate the impact of their full-featured environment against a stripped-down control environment, which does not allow for independent testing of the back-stepping feature.

Implementation of reverse debugging. Reverse debuggers are known for their high memory usage and challenging implementation. Programs are meant to be executed *forward*, destroying information and causing side effects in the process. In practice, reverse debuggers give the illusion to the programmer that the program executes in reverse by keeping a trace of the programs execution [10] and sometimes re-executing part of it [14]. Careful and efficient implementations are needed to provide programmers with a believable illusion.

Often, anecdotal evidence [10, 14], or the lack of alternative tools to solve a specific problem [14] are what justify the development of reverse debuggers. For instance, `RR`[14], developed by Mozilla, was developed to fix irreproducible bugs in the Firefox test suite. It uses as *record-and-replay* strategy: By recording all side effects of a program, it replays it to find the root cause of the bug. A record-and-replay tool is easily extended to a back-stepping debugger, by replaying the execution to a previous state, but it is not its primary goal.

Studying reverse-debugging lies at the intersection of multiple fields. On one hand, the expected benefits of free navigation between the past and the future of a program are attractive to the field of education in computer science. On the other hand, the significant implementation challenges of reverse debugging attract experts from programming language and debuggers. This could explain why back-stepping seems understudied from a human point of view.

3 User study

To evaluate the usefulness of back-stepping, we ran a within-subjects, mixed-methods user study with three debugging conditions: control (`C`), forward stepping only (`F`) and Both Forward and Back-stepping (`B`). Our goals were broadly focused on understanding the impact of the back-stepping feature on participants' debugging performance and strategy, addressing the following research questions:

(1) Does the back-stepping feature help programmers augment debugging success?
(2) Do programmers believe that forward and backward stepping are useful when debugging?
(3) What differences in strategies can be observed when different stepping features are available?

The following hypotheses were formulated. To answer them, we used the screen recordings, task completion time using the traces generated by codeBoot, NASA TLX cognitive load scale [4], and Likert-based perceived success ratings:

H1. Participants perceive tasks as less mentally demanding when using forward and back-stepping features.
H2. Participants rate debugging tasks as more successful when using back-stepping compared to other conditions.
H3. Participants perceive debugging interfaces with back-stepping as more helpful.
H4. Participants complete debugging tasks more successfully when using both forward and back-stepping features, compared to forward stepping alone or no stepping (control).

3.1 Study design and methodology

The study procedure gave time to participants to familiarize themselves with the back-stepping feature through a hands-on tutorial and an unconstrained pre-task period. Each session lasted approximately 75 minutes and a maximum of 90 minutes. After obtaining informed consent and completing a brief demographic questionnaire, the study procedure was as follows:

(1) **Tutorial of codeBoot interface:** Participants were given a 10 minute hands-on tutorial introducing the codeBoot interface, debugging features, and back-stepping feature.
(2) **Pre task:** Before each task, participants practiced the assigned condition using a sample task (Appendix 4), allowing them to explore the debugging features without time pressure.
(3) **Main Study Tasks:** Participants completed three debugging tasks, with each task performed under one condition. The order of the tasks was counterbalanced using a Latin square design to control for order effects, resulting in 18 unique orderings. Participants were encouraged to think aloud during this time while their screen, interactions and audio were recorded.
(4) **Post-task Questionnaire:** After each task, participants answered a short questionnaire about their experience with the debugging features (NASA-TLX).
(5) **Semi-structured Interview:** After all tasks, we ran a semi-structured interview to discuss participants' overall experience, perceptions of the debugging features, and their strategies.

3.2 Task design

To measure the impact of back-stepping, we created 3 different programs containing 4 bugs each. All bugs were biased towards back-stepping, requiring information on previous state of the program to be solved. To avoid confusion, participants were given a short test suite and were instructed to make all test cases pass. Test cases were written specifically to trigger bugs in the program. Tasks favored uncommon implementations: a pairwise mini-interpreter over expression lists (T1), a BFS-based frontier graph distance algorithm (T2), and a quicksort algorithm over Python dictionaries (T3). All tasks are included in Appendix A.

3.3 Recruitment and Participants

We recruited 18 participants for our study through targeted outreach in computer science and engineering contexts. Participants reported considerable programming experience ($\mu = 8.94$, $\sigma = 7.38$ years) and Python experience ($\mu = 4.11$, $\sigma = 1.75$ years). They showed regular engagement with debugging tools, with 8 using them almost every day and 9 using them at least once a week. The most popular debugging features were breakpoints and stepping through code, used by 94.4% of participants. Python and Java were among the most commonly used programming languages, with 14 participants reporting Python as one of their primary languages. Participants demonstrated a diverse programming toolkit, including JavaScript, C++, and Ruby.

Participants were distributed across age groups: 2 were between 18–22, 12 were between 23–27, 2 were between 28–34, and 2 were aged 35 or above. The gender distribution was 4 female and

14 male participants. Most participants (16 out of 18) were from computer science or engineering backgrounds. They were compensated $20 (CAD) for their time and contributions.

3.4 Data Analysis

Participant interactions were screen and audio recorded, with additional log data collected for debugging activities, including the number of steps taken, breakpoints set, and runs executed. All debugging files generated during the tasks were saved for later analysis.

We analyzed qualitative data using an inductive thematic analysis approach. Two coauthors performed affinity diagramming analysis from post-study interviews to derive clusters (codes), then met to discuss and resolve discrepancies until reaching consensus. The merged cluster set was then iteratively expanded as additional participant data were incorporated, continuing until thematic saturation was reached.

Quantitative data were analyzed using a repeated measures linear mixed effects model in `R` [11], examining the fixed effects of Condition, Task, and Order, along with all interaction effects between these factors, while controlling for the random effect of Participant. P-values for main and interaction effects were calculated using Satterthwaite's method for degrees of freedom, as implemented in the `lmerTest` package [7], and reported from ANOVA tables. Post-hoc comparisons were conducted using estimated marginal means (`emmeans`) with Bonferroni correction. When reporting estimates (β) and t-statistics (t), p-values are derived from the post-hoc tests.

We ensured that assumptions of normality (via Q-Q plots) and homoscedasticity were met by visually inspecting model residuals. Quantitative findings are intended to complement the qualitative insights by providing statistical evidence of the effects of Condition, Task, and Order on participants' performance and perceptions.

3.5 Analysis of debugging success

Because participants can make tests pass without solving the root cause of bugs, we measured debugging success in two ways: *test-passing success* and *expert-rated success* .

To measure *test-passing success*, each time a participant ran the code, a copy of the whole program was collected for analysis. All programs collected were executed against the test suite at the bottom of the file and successful resolution was attributed when a file passed the specific test.

To measure *expert-rated success*, the video footage was analyzed by the first author. Successful resolution of one of the 4 bugs was only attributed when the participant fixed the root cause of the bug, meaning that the fix prevents the program to misbehave in all intended uses of it. Successful attribution was accorded even if the participants later reintroduced the bug.

3.6 Implementation

We implemented reverse debugging on top of the codeBoot online environment with a record-and-replay approach similar to that of `RR` [14]. Through the execution of a program, the output of nondeterministic operations (such as I/O) is stored in a cache, and the interpreter keeps track of an execution step counter. In codeBoot, all expressions count as an execution step, hence there typically is more than one step per line of code [1]. When the user clicks on the back-step button, the program restarts from the beginning with all calls to nondeterministic operations replaced by a lookup in the cache, until the execution reaches the previous step. For long executions, this approach can add a perceivable delay when stepping back. However, the debugging tasks were short enough that backward steps were nearly instantaneous.

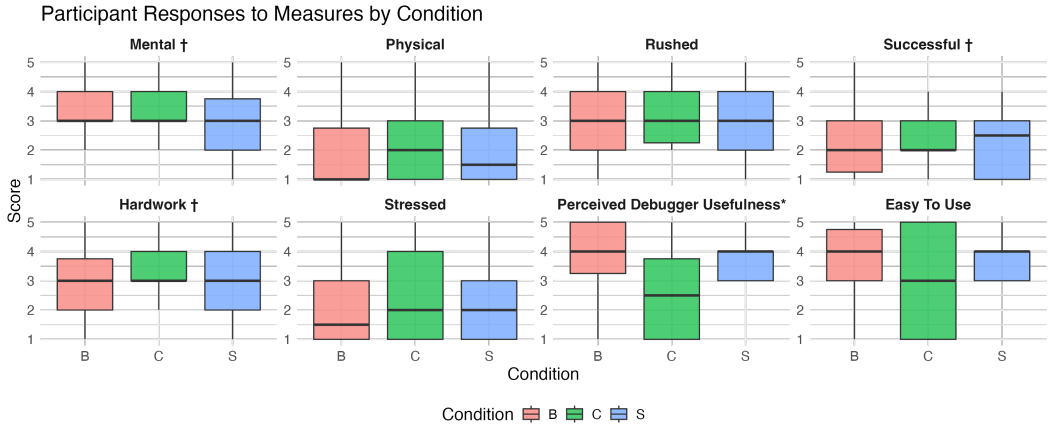

Fig. 2. Participant responses to Likert Questions for NASA TLX and system usability, grouped by Condition. Asterisk (*) indicates a significant main effect of Condition at $p < 0.05$; † indicates another main effect, interaction effect or outliers.

4 Findings

4.1 Quantitative results

Overall, participants perceived greater success in later task orders. With both forward and back-stepping support, the debugger improved participants' ability to identify bugs. Tasks also differed significantly in mental demand, with one task standing out as particularly challenging. No other questions reach significance with $p<0.05$.

4.1.1 ***Mental Demand (H1).*** We found a significant main effect of Task on perceived mental demand ($F(2, 25.993) = 3.70, p = 0.038$). Post-hoc pairwise comparisons with Bonferroni adjustment revealed that Task T1 was perceived as significantly less mentally demanding than Task T2 ($\beta = -0.967, t = -5.463, p = 0.0001$), and that Task T2 was significantly more mentally demanding than Task T3 ($\beta = 0.556, t = 3.298, p = 0.0103$). These results indicate that Task T2 imposed the greatest cognitive workload, while Task T1 was perceived as the least demanding.

4.1.2 ***Perceived Successfulness (H2).*** For perceived successfulness, we observed a significant main effect of Order ($F(2, 12.185) = 10.39, p = 0.002$). Participants reported feeling significantly less successful in Order 1 compared to Order 3 ($\beta = -1.133, t = -4.400, p = 0.0021$), and also less successful in Order 2 compared to Order 3 ($\beta = -0.774, t = -3.181, p = 0.0227$). These findings suggest that participants' perceived success improved over time, consistent with a learning or adaptation effect across task orders.

4.1.3 ***Perceived Debugger Usefulness (H3).*** We found a significant main effect of Condition on participants' perceptions of how the debugging interface helped them identify bugs ($F(2, 16.074) = 8.14, p = 0.003$). Post-hoc comparisons with Bonferroni correction showed that participants rated Condition B (forward stepping with back-stepping) significantly higher than Condition C (no debugging) ($\beta = 1.557, t = 3.696, p = 0.0075$). Condition S (forward-only stepping) was also rated significantly higher than Condition C ($\beta = -1.331, t = -3.237, p = 0.0191$). Overall, these results suggest that the B condition provided the most effective perceived debugging support, followed by the S condition, while the C condition offered the least support for identifying bugs.

4.1.4 ***Test-passing success (H4)***. A significant main effect of Task was found ($F(2, 28.00) = 5.67, p = 0.0085$). No significant main effects were observed for Condition ($F(2, 28.00) = 0.21, p = 0.8133$) or Order ($F(2, 28.00) = 1.93, p = 0.1646$). The Condition × Task interaction did not reach significance ($F(4, 21.43) = 2.18, p = 0.1059$). Post-hoc analysis of Task showed: Task T1 was completed significantly less often than Task T3 ($\beta = -0.667, t = -3.353, p = 0.0063$), while the difference between Task T2 and Task T3 did not reach significance ($\beta = -0.389, t = -1.956, p = 0.1421$). Although interaction effects were not statistically significant, further post-hoc analyses indicated that in the Control condition, participants solved significantly more bugs in Task T3 than in Tasks T1 and T2 (both $\beta = -1.500, t = -2.700, p = 0.0315$). This pattern suggests that task characteristics, rather than debugging condition alone, primarily drove objective completion outcomes.

4.1.5 ***Expert-rated Success (H4)***. A significant main effect of order (F(2, 12.10) = 11.23, p = 0.0017) and Task (F(2, 12.10) = 9.85, p = 0.0029) was found. Condition almost reached significance (F(2, 12.10) = 3.84, p = 0.0512). Only interaction effects between Order and Task approached significance (F(4, 15.56) = 2.58, p = 0.0787). Post-hoc analyses showed that Task T1 was completed significantly less successfully than Task T3, with a medium effect size ($\beta = -0.667, t = -3.518, p = 0.0109, d = -0.70$), and that Task T2 was also completed significantly less successfully than Task T3, again with a medium effect size ($\beta = -0.778, t = -4.104, p = 0.0038, d = -0.70$).

For order, participants performed significantly better in Order 3 than in Order 1, with a large effect size ($\beta = 0.889, t = 4.690, p = 0.0014, d = 0.86$),and significantly better in Order 2 than in Order 1, with a medium effect size ($\beta = 0.556, t = 2.931, p = 0.0313, d = 0.50$).

Although not statistically significant, expert ratings suggested that participants in the Back-stepping condition tended to perform better than those in the Stepping condition, with a medium effect size ($\beta = 0.500, t = 2.638, p = 0.0526, d = 0.50$). Participants in the Control condition also appeared to perform slightly better than those in the Stepping condition, though this effect was small and did not reach significance.

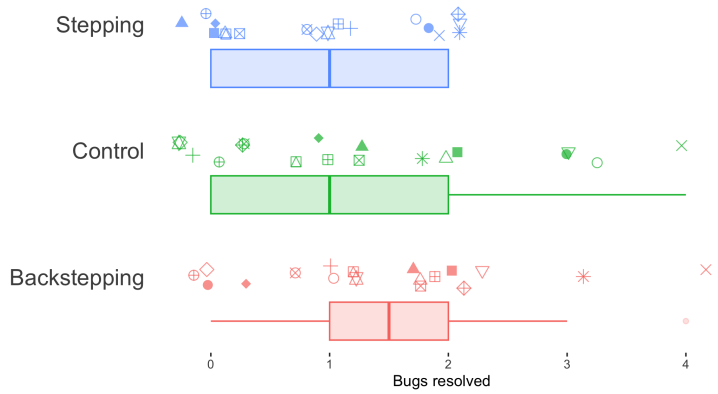

Fig. 3. Participants number of bugs resolved for each condition. Each shape shows a different participant.

Overall, Task 2 was significantly less completed with participants finding 2/3 less bugs on average than the other tasks. Tasks performed last were significantly more completed with participants

finding 0.89 and 0.56 additional bugs on average compared to the first and second task. Approaching significance ($p = 0.0526$), participants using the back-stepping feature found 0.5 more bugs compared to using stepping as shown by Figure 3.

4.2 Qualitative Results

4.2.1 Back-stepping removes the fear of "missing the moment" and reduces cognitive load by allowing reversible investigations. When back-stepping was unavailable, participants described debugging as fragile: stepping past a critical state often meant losing access to relevant information and being forced to restart execution. This created what participants described as a fear of "missing the moment" (P6), encouraging cautious, slow navigation to avoid overshooting informative execution points: "As soon as you arrive at the right place but you pass it, you have to relaunch everything" (P6).

Back-stepping reduced this cost by allowing participants to revisit previously observed states without restarting the program or rewinding execution in their mind. This enabled a more flexible investigative style, where participants could overshoot, backtrack, and directly inspect how control flow and state changed over time. One participant explained that back-stepping eliminated the need to mentally trace conditionals: "I have to look at [all the if conditions] and tell myself 'it wouldn't have entered that one.' [With step-back] I could just go back and see which if it enters." (P9).

Yet, some participants raised concerns that reducing the fear of "missing the moment" could encourage less deliberate reading of code. As one participant reflected, "If it's too easy to go back, maybe you'll go faster—and it might even take more time to find bugs" (P1).

4.2.2 Reverse debugging shifts strategies toward failure-first navigation and backtracking. When back-stepping was available, participants' debugging strategies shifted from careful, anticipatory navigation toward a failure-first approach: rather than planning breakpoints or stepping to intercept errors, participants more readily ran programs until failure and then worked backward to locate the cause. One participant described this shift as a reversal of their usual debugging direction: "I started going straight to the error right away, and then doing step-backs… it's like I'm going backward instead of going forward" (P2).

This contrasted strongly with what participants did when reverse execution was unavailable. Without back-stepping, participants described modifying code to insert print statements or carefully placing breakpoints in advance, often resulting in cleanup overhead and disrupted flow. As one participant explained, "You have to modify the code, restart execution… I tend to add tons of prints, and after that you have to go clean up" (P9).

Although failure-first navigation was often perceived as faster and more direct, participants also noted that because they could always rewind, they spent less time understanding the code up front. As one participant reflected, "When I had the debugger on hand, I spent less time at the beginning going through the code… I let myself be guided more by the debugger" (P8).

4.2.3 The lack of visual feedback for navigation inside debuggers makes back-stepping hard to master. Participants described back-stepping as introducing an additional temporal dimension to debugging, which made navigation harder to grasp and increased the risk of disorientation: P3 described this difficulty as simply "being too lost" to know when back-stepping would help. Its benefits were not immediately accessible due to a "learning curve period" (P8).

Participants said that part of the challenge came from not being able to see more than the current line. They wanted some visual cue about the surrounding execution context—such as where they were in the call stack or what the next jump would be—to better predict how stepping would behave. As one participant put it, "Just materializing the stack somewhere visually would help… for orientation in the code execution" (P4).

Others wished for a preview showing what a leap or step-back would actually do, noting that without this, they often overshot or hesitated. “Sometimes it’s hard to know the limit,” one participant explained when talking about leap and back-leap behavior (P6).

4.2.4 Back-stepping is seen as a general feature that can replace and complement many others. As participants grew comfortable with back-stepping, several began to see it not as a single debugging tool, but as a general capability that could substitute for, or enhance, many existing features. They described it as a way to instantly regain access to past program states, something normally achieved through stack traces, breakpoints, or manually inspecting variables. P2 captured this feeling by calling back-stepping a “cheat code” because it shows “the value of all the variables at the moment the error happens” (P2).

Participants compared back-stepping to tools they normally rely on to approximate the past. For example, stack traces or “set next statement” gave only partial visibility, whereas reverse execution allowed them to truly move to earlier points with the correct state intact. As one participant explained, “I can simulate stepping forward with breakpoints… but there is no way to step back with breakpoints” (P14).

Back-stepping is also viewed as a feature that can be paired with established features. P6 framed back-stepping as complementary to live state inspection: "I would say a combination of backward return and watch [displayed values], really having the values live [help most when debugging]" (P6), while also noting that richer error-time state alone can sometimes reduce the need to rewind: “It’s good to have both. But if you already see on your bug what your values are, you aren’t forced to start to back step.” (P6). P13 combined it with breakpoints as a routine workflow:" I used the breakpoints and the step-back tools ..." (P13)

5 Discussion

Our results show that back-stepping is rated as more useful than stepping by users, facilitates the investigation of bugs by removing the fear of “missing the moment”, and changing the overall debugging strategy by prioritizing failure-first navigation and backtracking. Analysis of expert-rated success showed an almost significant effect of back-stepping compared to stepping ($p = 0.0526$), corresponding to an average increase of 0.5 bugs resolved (out of four).

Although this suggests that back-stepping may help participants resolve more bugs, this effect was not observed when comparing against the control condition or when success was measured using unit tests. These differences are further discussed in the limitations section.

Our findings provide grounds to argue that back-stepping is not detrimental to debugging success. This contrasts with results from Thayer, Guo and Reinecke [18], who found a negative correlation between debugging success and the usage of the back-stepping button. A key difference between the studies is participants’ experience: Thayer, Guo and Reinecke recruited novice programmers whereas we recruited experienced programmers. This suggests that the back-stepping button as an interface to reverse debugging might be less suited to novice programmers but beneficial for experienced ones. This interpretation is consistent with the qualitative feedback indicating that experienced programmers have trouble understanding their current position in the execution, suggesting that this effect should be more pronounced for novices.

Another difference between Thayer, Guo and Reinecke’s study [18] and ours is the nature of the programming tasks. Our tasks are more challenging and deliberately biased towards back-stepping, suggesting that back-stepping’s effectiveness is situational. This is consistent with our qualitative findings indicating that back-stepping is more useful when bugs need uninterrupted investigations to be solved or when working backwards from a clear crashing state, conditions not present in Thayer, Guo and Reinecke’s study.

Overall, our study highlights the need for debugging features like back-stepping, as we show its utility in helping participants in specific situations. These findings contribute to a body of research on reverse debugging by providing evidence that back-stepping can enhance debugging success when appropriately integrated into programmers' workflows.

Future work should explore long-term adaptation, novice programmer experiences, and further refinements to the interface design, ensuring that tools like back-stepping strike a balance between enhancing efficiency and fostering deep code comprehension.

5.1 Undo and redo as an interface to reverse debugging?

We observed that during debugging, participants wanted to revisit the program state they had previously seen, and used the back-stepping and forward-stepping buttons trying to get back to that state. This is similar to trying to manually reconstruct an earlier version of a file, remembering the words and phrases present before the changes. Using this comparison, we believe that presenting reverse debugging as a history of the user's path within the program could provide a more intuitive mental model, as it avoids the extra burden of remembering this path. This could be implemented by retaining all actions taken by the user (stepping forward, breakpoint stops, etc) and applying the reverse actions on demand, sometimes triggering the reverse debugger to "step back" from the last executed action. The history doesn't have to be implemented using undo/redo buttons, although this interface would be the simplest.

This interface is consistent with our qualitative finding that debugging incurs a fear of "missing the moment", as access to the history would reassure the user that any missed state can be recovered. This could also help users situate themselves in the dynamic execution context, something that participants struggled with. However, additional evidence will be needed to conclude on this promising new way of viewing reverse debugging.

5.2 Limitations

Our within-subjects study, while efficient for our sample size, has limitations. First, despite using a counterbalanced Latin square design, a strong learning effect persisted: participants improved with the debugging interface over time, potentially confounding results. Additionally, Task 2 was notably more challenging, as reflected in both quantitative workload data and qualitative feedback, which may have skewed performance outcomes. This may help explain the difference in observation between the expert-rated and test-passing success measures, as we saw that participants had more trouble resolving bugs in Task 2 than in other tasks. Second, usability issues, such as challenges with viewing code while stepping because the floating variable box obscured portions of the editor-emerged in qualitative feedback and may have hindered efficient interaction with the debugging tools. This can partly explain the lack of observable differences between the control condition and the stepping condition in expert-rated success. Third, the controlled environment with simplified debugging tasks limits the generalizability of our findings to real-world scenarios. Both quantitative data and qualitative interviews suggest that the tool's effectiveness may vary with task complexity and developers' workflows. Moreover, our focus on experienced developers restricts applicability to novice programmers, who may have different needs and strategies. Finally, the short-term study design prevented us from observing long-term adaptation to forward and backward stepping. While participants qualitatively reported the transformative potential of these features, their long-term impact-such as over-reliance or shifts in debugging strategy requires further investigation during extended use.

References

[1] Marc Feeley and Olivier Melançon. 2022. Teaching Programming to Novices Using the codeBoot Online Environment. *Electronic Proceedings in Theoretical Computer Science* 363 (July 2022), 44–53. doi:10.4204/EPTCS.363.3 arXiv:2207.12702 [cs].

[2] GNU Project. 2024. *GDB: The GNU Project Debugger*. Free Software Foundation. https://www.gnu.org/software/gdb/documentation/ Version 15.2.

[3] Philip J. Guo. 2013. Online Python Tutor: a web-based program visualization tool for teaching programming. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. ACM, 579–584. https://pg.ucsd.edu/publications/Online-Python-Tutor-web-based-program-visualization_SIGCSE-2013.pdf

[4] SG Hart. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. *Human mental workload/Elsevier* (1988).

[5] Haskell.org. 2023. *The GHCi Debugger*. Haskell.org. Accessed: 2024-12-16.

[6] Simon Peyton Jones. 2010. *Haskell 2010 Language Report*. Language Report. Haskell Committee. https://www.haskell.org/definition/haskell2010.pdf

[7] Alexandra Kuznetsova, Per B Brockhoff, and Rune Haubo Bojesen Christensen. 2017. lmerTest package: tests in linear mixed effects models. *Journal of statistical software* 82, 13 (2017).

[8] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2024. *The OCaml System Release 5.10*. INRIA. Accessed: 2024-12-16.

[9] Demian Lessa, Bharat Jayaraman, and Jan Chomicki. 2011. Temporal Data Model for Program Debugging. In *Database Programming Languages - DBPL 201, 13th International Symposium, Seattle, Washington, USA, August 29, 2011. Proceedings*, Nate Foster and Anastasios Kementsietsidis (Eds.). https://research.cs.cornell.edu/conferences/dbpl2011/papers/dbpl11-lessa.pdf

[10] Bil Lewis and Bil Lewis. 2003. Debugging Backwards in Time. *arXiv: Software Engineering* (2003). doi:null

[11] Steven G Luke. 2017. Evaluating significance in linear mixed-effects models in R. *Behavior research methods* 49 (2017), 1494–1502.

[12] Microsoft. 2024. IntelliTrace. https://learn.microsoft.com/en-us/visualstudio/debugger/intellitrace?view=vs-2022. Accessed: 2024-12-16.

[13] Microsoft Visual Studio Blog. 2024. Introducing Time Travel Debugging for Visual Studio Enterprise 2019. https://devblogs.microsoft.com/visualstudio/introducing-time-travel-debugging-for-visual-studio-enterprise-2019/. Accessed: 2024-12-16.

[14] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record And Replay For Deployability: Extended Technical Report. arXiv:1705.05937 [cs.PL] https://arxiv.org/abs/1705.05937

[15] OCaml.org. 2023. *OCaml Debugger*. OCaml.org. Accessed: 2024-12-16.

[16] RevDeBug. 2024. RevDeBug. https://revdebug.com/. "Accessed: 2024-12-16".

[17] Stack Exchange contributors. 2023. Why is reverse debugging rarely used? https://softwareengineering.stackexchange.com/questions/181527/why-is-reverse-debugging-rarely-used. Accessed: 2024-12-16.

[18] Kyle Thayer, Philip J. Guo, and Katharina Reinecke. 2018. The Impact of Culture on Learner Behavior in Visual Debuggers. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 115–124. doi:10.1109/VLHCC.2018.8506556

[19] Undo.io. 2024. UndoDB. https://undo.io/products/udb/. Accessed: 2024-12-16.

A Debugging Tasks

The debugging tasks were designed to reflect real-world scenarios where back-stepping could provide a clear advantage. Each task triggered an exception when first executed, intentionally favoring back-stepping through its ability to inspect the pre-crash program state. Subsequent bugs where difficult to see through direct code inspection, needing intermediate debugging information to be solved. Bugs increased in difficulty for each tasks. Multiple pilot studies were conducted to asses task difficulty and calibration.

T1 Mini-interpreter. A simple evaluator for mathematical expressions containing only + and - operators. The evaluation function was uncommon, looping through a list of operands and operator pair-by-pair to compute the result.

T2 Graph distance algorithm. We used the a BFS frontiers algorithm, meaning that at each iteration all the neighbors from all the nodes were added to the open list. This allowed for complex loop logic, having three nested loops to compute the distance.

T3 A quick-sort algorithm. This task deviated from the classical recursive quick-sort algorithm, sorting dictionaries by a key given as an argument.

A.1 Task 1: Mini-interpreter

No Bugs Version

```
NUMBERS = "0123456789"
def split_expressions(string):
    expressions = []
    i = 0
    while i < len(string):
        c = string[i]
        if c in NUMBERS:
            acc = int(c)
            i += 1
            while i < len(string) and string[i] in NUMBERS:
                acc = acc * 10 + int(string[i])
                i += 1
            i -= 1
            expressions.append(acc)
        elif c == "+" or c == "-":
            expressions.append(c)
        i += 1
    return expressions

def evaluate_expressions(expressions):
    current_value = expressions[0]
    rest = expressions[1:]
    for i in range(len(rest) // 2):
        operator = rest[i*2]
        next_value = rest[i*2+1]
        if operator == "+":
            current_value += next_value
        elif operator == "-":
            current_value -= next_value
    return current_value

def evaluate(string):
    return evaluate_expressions(split_expressions(string))
# Tests
def test(result, expected):
    if result == expected:
        print("PASS")
    else:
        print("FAIL (got", repr(result), "expected", repr(expected), ")")

print("Tests will be displayed after this line")
test(evaluate("1"), 1)
test(evaluate("1 + 2 + 3"), 6)
```

```
test(evaluate("45 + 1"), 46)
test(evaluate("1-2-3"), -4)
```

Listing 1. Task 1: Evaluate Mathematical Expressions

A.2 Task 2: Graph Distance Calculation

No Bugs Version

```
def create_graph():
    return dict()

node_counter = 0
def add_node(G):
    global node_counter
    node = node_counter
    G[node] = []
    node_counter += 1
    return node

def add_edge(G, node_1, node_2):
    G[node_1].append(node_2)
    G[node_2].append(node_1)

def distance(G, node_a, node_b):
    queue = []
    seen = [node_a]
    neighbours = [node_a]
    dist = 0

    while len(neighbours) != 0:
        next_neighbours = []
        for current_node in neighbours:
            for neighbour in G[current_node]:
                if neighbour not in seen:
                    next_neighbours.append(neighbour)
                    seen.append(neighbour)
            if current_node == node_b:
                return dist
        dist += 1
        neighbours = next_neighbours
    return -1

# Tests
def test(result, expected):
    if result == expected:
        print("PASS")
    else:
        print("FAIL (got", repr(result), "expected", repr(expected), ")")

print("Tests will be displayed after this line")
G = create_graph()
n1 = add_node(G)
```

```
test(distance(G, n1, n1), 0)

n2 = add_node(G)
test(distance(G, n1, n2), -1)

add_edge(G, n1, n2)
test(distance(G, n1, n2), 1)

n3 = add_node(G)
add_edge(G, n2, n3)
test(distance(G, n1, n3), 2)
```

Listing 2. Task 2: Graph Distance Calculation

A.3 Task 3: Quick Sort Implementation

No Bugs Version

```
def quick_sort(objs, obj_key):
    if len(objs) < 2:
        return objs
    else:
        pivot_index = len(objs) // 2
        left, right = partition(objs, obj_key, pivot_index)
        pivot_value = objs[-1]
        return quick_sort(left, obj_key) + [pivot_value] + quick_sort(
            right, obj_key)

def partition(objs, obj_key, pivot_index):
    temp = objs[-1]
    objs[-1] = objs[pivot_index]
    objs[pivot_index] = temp
    left = []
    right = []
    pivot_value = objs[-1][obj_key]
    for elem in objs[:-1]:
        if elem[obj_key] <= pivot_value:
            left.append(elem)
        else:
            right.append(elem)
    return left, right

# Tests
def test(result, expected):
    if result == expected:
        print("PASS")
    else:
        print("FAIL (got", repr(result), "expected", repr(expected), ")")

print("Tests will be displayed after this line")
objs = [{'value': 2}, {'value': 1}, {'value': 1}]
test(quick_sort(objs, 'value'), [{'value': 1}, {'value': 1}, {'value':
    2}])
```

```
34
35 objs = [{'amount': 0}, {'amount': 0}, {'amount': 1}, {'amount': 0}, {'
       amount': 0}, {'amount': 0}]
36 expected = [{'amount': 0}, {'amount': 0}, {'amount': 0}, {'amount': 0}, {
       'amount': 0}, {'amount': 1}]
37 test(quick_sort(objs, 'amount'), expected)
38
39 objs = [{'dollars': 3}, {'dollars': 2}, {'dollars': 1}]
40 test(quick_sort(objs, 'dollars'), [{'dollars': 1}, {'dollars': 2}, {'
       dollars': 3}])
41
42 objs = [{'things': 1}, {'things': 5}, {'things': 2}, {'things': 4}, {'
       things': 2}]
43 expected = [{'things': 1}, {'things': 2}, {'things': 2}, {'things': 4}, {
       'things': 5}]
44 test(quick_sort(objs, 'things'), expected)
```

Listing 3. Task 3: Quick Sort Implementation

A.4 Sample task

```
1  # [EN] Execute the code to see the first bug
2  # [FR] Execute le code pour voir la premiere erreur
3
4  def add(x, y):
5      # [EN] Bug here
6      # [FR] Bogue ici
7      return x + y + z
8
9  def mult(x, y):
10     acc = 0
11
12     for i in range(y):
13         # [EN] Bug here: 'y' should be 'x'
14         # [FR] Bogue ici: 'y' devrait etre 'x'
15         acc = add(acc, y)
16
17     return acc
18
19 # [EN] Can you find the bug in this code?
20 # [FR] Peux-tu trouver le bogue dans ce code?
21 def power(x, y):
22     acc = 0
23
24     for i in range(y):
25         acc = mult(acc, x)
26
27     return acc
28
29 # [EN] No bugs after this line
30 # [FR] Pas de bugs apres cette ligne
31
```

```
def test(result, expected):
    if result == expected:
        print("PASS")
    else:
        print("FAIL (got", repr(result), "expected", repr(expected), ")")

print("Tests will be displayed after this line")
test(add(2, 3), 5)
test(mult(2, 3), 6)
test(power(2, 3), 8)
```

Listing 4. Presented before each task with the tested condition available.

accepted 16 December 2024