Accepted Manuscript, to appear in:
UK and Ireland Computing Education Research 2026 (UKICER 2026)

Towards a Theory-Based Approach for Analyzing Students' Debugging Processes

Elena Spörer
elena.spoerer@tum.de
Technical University of Munich
Computing Education Research Group
Munich, Germany

Tilman Michaeli
tilman.michaeli@tum.de
Technical University of Munich
Computing Education Research Group
Munich, Germany

Abstract

Debugging is an immense challenge for novices when learning programming, often causing frustration. Applying a systematic debugging process has a considerable impact on debugging success. As a result, several interventions have been designed to teach this process to students. However, understanding how students actually apply these processes remains limited, as existing analysis approaches only partially capture students' debugging processes. To address this gap, this paper proposes a process-oriented and programming language-independent analysis approach grounded in an ideal-typical debugging process model to reconstruct and analyze debugging processes. This approach provides a foundation for process-level insights into how students identify and solve bugs. We illustrate the applicability of the approach through a case study analyzing four screen recordings of secondary school students debugging Scratch programs. First insights demonstrate how the approach enables a detailed reconstruction of debugging processes and the identification of deviations from an ideal-typical debugging process, such as missing verification steps.

CCS Concepts

• **Social and professional topics → K-12 education**.

Keywords

debugging, process analysis, secondary school, theory-based, computing education

ACM Reference Format:
Elena Spörer and Tilman Michaeli. 2026. Towards a Theory-Based Approach for Analyzing Students' Debugging Processes. In *UK and Ireland Computing Education Research 2026 (UKICER 2026), September 03–04, 2026, Cambridge, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3830800.3830802

1 Motivation

Debugging is an unavoidable yet persistently challenging aspect of learning to program, often causing frustration [21]. A key factor for successful debugging is the application of an effective process which involves formulating and verifying hypotheses [28]. Accordingly, various interventions have been developed to support students in learning a systematic debugging process, which have been shown to successfully improve debugging performance [19, 26].

To understand why and how these interventions work and how they can be designed effectively, as well as to identify where students struggle, analyzing students' debugging processes is essential. Therefore, previous research has approached the analysis of debugging processes in various ways. One common approach is the use of program snapshots to analyze students' debugging processes. However, they only provide partial insights into students' activities, as they omit all steps performed between two snapshots [1, 18]. Another line of research has applied inductive approaches to identify recurring patterns, primarily focusing on debugging strategies and performance [24, 25]. While these approaches provide valuable, context-specific insights, they limit comparability both within and between learners. In contrast, other studies adopt theory-based approaches grounded in an ideal-typical debugging process model to analyze individual debugging steps [16, 20]. While those approaches facilitate more systematic comparisons, they typically do not consider the sequential structure of the steps within the debugging process, thus missing important process-oriented insights into how students work. In summary, we argue that this limits our understanding of students' debugging processes and misses potential to diagnose students' problems, explain unsuccessful debugging attempts, and design targeted interventions [22].

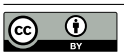
To this end, we propose a theory-based analysis approach that reconstructs students' debugging processes by mapping observable actions to the steps of an ideal-typical debugging process model. The approach uses screen recordings of students working on debugging tasks as its primary data source. Specifically, this paper contributes (1) a theory-based approach for a fine-grained reconstruction and analysis of debugging processes, and (2) an illustrative case study demonstrating its applicability.

2 Related Work

Debugging, the systematic process of finding and fixing bugs, is an essential part of software engineering as most programs do not work correctly on the first attempt. Previous research has focused on various aspects of debugging, including typical bugs in novices' programs [3, 4, 7], and approaches to teach debugging [6, 23]. In the following, we first outline models that define the ideal-typical debugging process in detail and then we review previous studies investigating students' debugging processes.

2.1 The Ideal-Typical Debugging Process

There are several models that describe the *ideal-typical* debugging process and specify the essential steps of a successful process. Those



UKICER 2026, Cambridge, United Kingdom

ACM ISBN 979-8-4007-2593-7/2026/09
https://doi.org/10.1145/3830800.3830802

models were either developed empirically [10, 11] or theoretically [2, 28]. Despite differences in their development and target group, the models can be summarized into the same four steps: (1) observe the failure, (2) set up a hypothesis, (3) verify the hypothesis, and (4) fix the bug and verify the solution.

Accordingly, this process is often called as *scientific debugging*, as the iterative identification and verification of hypotheses is a central part of the process. Most debugging process models focus on localizing the bug by describing the steps (1) to (3) in detail (e.g., [10, 11, 28]). However, fixing the bug (step (4)) can also be a considerable challenge, especially for novices [23]. Accordingly, some models elaborate on this step in more detail. For example, Araki et al. (1991) distinguish between two overarching phases: locating bugs and correcting bugs (see Figure 2a). By introducing a dedicated phase for fixing bugs, which contains more than one step addressing the bug fix, the importance of this phase within the debugging process is emphasized. Compared to other process models, this results in a more fine-grained description of the ideal-typical debugging process.

While this highlights how process models can differ in their level of detail, another important distinction concerns the context of debugging. In particular, it is important to distinguish between debugging one's own code and debugging code written by others, as research shows these situations require different approaches [12, 14]. When debugging unfamiliar code, additional code comprehension steps are necessary [14] and should therefore be added to the ideal-typical debugging process. However, only a few models explicitly account for these comprehension steps (e.g., [10, 27]).

2.2 Analyzing Debugging Processes

A recent review by Spörer and Michaeli (2025) summarized the methodologies that were used to investigate debugging processes in computing education research. They found that studies analyzing debugging processes typically rely on qualitative methodologies and investigate either videos (from the classroom or the students' screens) [13, 20, 24] or program snapshots [1, 18]. These studies primarily aim to describe cognitive aspects of debugging, such as the usage of debugging strategies [8, 24] and the students' debugging performance [15, 17].

As a foundation for assessing students' debugging processes, previous research used a vast variety of data types, e.g., videos [13, 20, 24], program snapshots [1, 18], interviews [16], and surveys [9]. These data sources differ regarding their information content and the effort required for analysis. While videos capture everything the students do and say and thus have the potential to record the entire debugging process, the analysis is highly time-consuming. In contrast, program snapshots provide only static copies of the code at certain points in time, omitting all steps performed between two snapshots and therefore capturing only fragments of the process, but enabling efficient and scalable automated analysis [22].

Focusing on data analysis, most studies used an inductive approach, primarily to identify debugging strategies used by the students [8, 24]. In contrast, studies adopting a deductive analysis approach have examined debugging-related aspects such as social regulation or self-regulation strategies and collaboration [13, 20]. Only a few studies [5, 16, 20] have explicitly analyzed students' debugging processes using debugging process steps as a theoretical framework for deductive analysis. For example, Parkinson et al. (2024) and Kim et al. (2018) investigated whether the process steps from an ideal-typical debugging process could be identified in the students' debugging processes. While prior work identifies individual debugging steps, it does not address how these steps are connected within the sequential structure of the debugging process. Building on this work, we argue that, since debugging is an iterative process, analyzing this sequential structure offers important insights into the underlying processes. This, in turn, enables a more detailed understanding of how students approach and resolve bugs.

Taken together, prior research provides several insights into specific aspects of students' debugging behavior but lacks approaches that capture and analyze the full sequential structure of debugging processes. In particular, theory-based approaches that map the students' processes to an established debugging process model are largely missing.

3 A Theory-Based Approach to Analyze Debugging Processes

To address this gap, we propose a theory-based analysis approach for investigating students' debugging processes. Our approach reconstructs debugging processes per individual error by mapping observable actions to the programming language-independent steps of an ideal-typical debugging process model. As systematic debugging processes are known to support successful debugging, we argue that comparing students' processes to an ideal-typical process provides a meaningful reference point for analysis. Moreover, such a process-oriented perspective on debugging is important for the identification of where and why students deviate from an ideal-typical process. We use screen and audio recordings as data sources, as they capture both students' interactions with the programming environment and their verbalizations. On top of that, screen recordings allow us to capture the entire debugging process.

To clarify the terminology used in our approach, we define the following central terms:

- An **action** is an observable behavior that occurs when students interact with a programming environment. Actions can be programming language-specific or programming language-independent.
  - Examples of programming language-specific (Scratch) actions are duplicating blocks, inspecting drop-down menus, and adding new blocks to the workflow.
  - Examples of programming language-independent actions are changing variable values, scrolling through the code, parsing the code, and executing the program.
- A **debugging episode** is the set of all actions that were carried out with the intention of solving one specific bug in a program.

Based on these definitions, our proposed approach follows four steps: (1) coding of observable actions, (2) grouping actions into debugging episodes, (3) mapping actions within each debugging episode to debugging process steps, and (4) comparing the mapping with the ideal-typical debugging process (refer to Figure 1).

Step 1: Coding of observable actions. The first step focuses on a descriptive coding of all observable actions visible in the screen recordings of the debugging processes. This results in a fine-grained overview of the students' actions and serves as the basis for the subsequent evaluation and interpretation steps. Audio recordings are not necessary for that step, as all actions can be directly observed from the screen recordings.

Coding actions from screen recordings offers the chance to derive detailed information. We deliberately chose to use screen recordings as data source, as they contain more contextual information and require less technical features than other approaches that provide fine-grained process descriptions like log files.

Step 2: Grouping actions into debugging episodes. In the second step, all actions related to the same error are grouped into the same debugging episode, regardless of when they occur in the screen recording. The audio data is used to support this assignment and ensure that actions are grouped correctly.

Grouping actions into debugging episodes allows the reconstruction of debugging processes for each error individually. This reveals whether students follow linear debugging processes, or whether different processes interrupt or overlap each other. In addition, this structuring allows comparisons of debugging processes across different types of errors.

Step 3: Mapping actions within each debugging episode to debugging process steps. Building on steps 1 and 2, the actions within each debugging episode are mapped to the steps of an ideal-typical debugging process model, which can be chosen from the models proposed in the literature. Depending on the context, one or more actions are mapped to the same process step. The audio tracks provide additional information about the students' approaches, especially in phases with no mouse movement.

This mapping enables the interpretation of observable behavior and the reconstruction of the students' debugging processes. It further provides a foundation for systematically comparing these processes to an ideal-typical debugging process and, by abstracting from language-specific actions, it facilitates the comparison of processes across learners, tasks, and contexts.

Step 4: Comparing mapping with the ideal-typical debugging process. In the final step, the students' debugging processes are visualized and compared to the ideal-typical debugging process.

By doing so, the comparison enables the identification of missing or additional steps within the debugging processes. Moreover, this process-oriented perspective reveals how and where students' debugging processes differ from a systematic process.

4 A Case Study Demonstrating the Proposed Approach

To demonstrate the potential of the proposed approach, we conducted a case study which serves as an illustrative example rather than as a basis for generalizable conclusions.

4.1 Context and Data Collection

The data was collected in the context of a five-day workshop on programming games, which targeted motivated secondary school students who signed up voluntarily. Therefore, this does not represent a typical classroom sample. Five students (two female and three male, grades 9 to 11) participated, each with one to three years of programming experience in the context of computing education lessons. On days two to four, the participants worked on short debugging tasks used as morning warm-up challenges aligned with the workshop topic. They did not receive any prior instruction on debugging or ideal-typical debugging processes. Informed written consent was obtained from all participants and their legal guardians before the workshop.

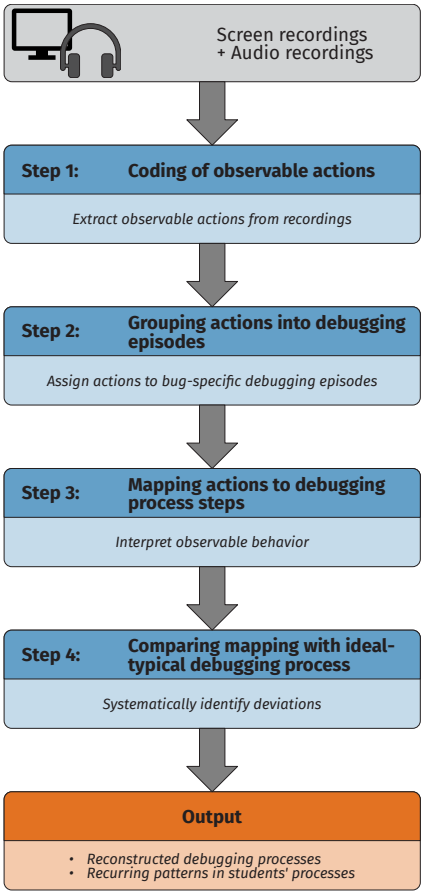


Figure 1: Visualization of the steps of the proposed approach.

The debugging tasks were based on Scratch implementations of the games Snake and Pong. All participants had previously worked with Scratch. Although the students were familiar with the general functionalities of the games, they had not seen the specific implementation before, so they were debugging others' code. Each debugging task was designed to represent four or five typical novice programming errors, which were derived from the literature (e.g., [3, 4, 7]). These errors were mostly programming language-independent, including incorrect variable assignments, incorrect conditional statements, and confusion between assigning and incrementing variable values. For each task, the students received a brief description about the expected program functionalities and had approximately 35-40 minutes to complete it.

The data collection approach was inspired by previous research from Parkinson et al. (2024) and Fitzgerald et al. (2008): To solve the debugging tasks, the students worked in pairs to encourage a natural conversation about their thoughts on the tasks. One participant completed the tasks individually due to the group size. Screen and audio recordings were collected for all sessions.

4.2 Application of the approach

To demonstrate the applicability of the proposed analysis approach, we analyzed four screen recordings of students working on debugging tasks. The other five recordings had to be excluded due to missing audio data. In the following, we first illustrate the application of the approach in detail using one screen recording from the case study before identifying recurring patterns across all cases. In the selected example two students (female; at the end of grade 9) are debugging a Scratch implementation of the game Snake. The task contained four predefined bugs the students were asked to identify and solve:

- **incorrect variable value**: the head of the snake looks in the wrong direction at the beginning of the game
- **incorrect variable value:** the head of the snake looks in the wrong direction during runtime
- **incorrect variable value**: the tail does not follow the head correctly
- **disconnected block**: the head of the snake is visible on the start screen

Step 1: Coding of observable actions. We inductively developed a coding scheme for observable actions based on the screen recordings. The resulting set of actions includes both programming language-specific actions (e.g., adding or deleting blocks) and programming language-independent actions (e.g., navigating the code, modifying variable values, or executing the program). The first author conducted the coding. As the coding of the actions was purely descriptive, we did not conduct an inter-coder reliability check for this step.

Step 2: Grouping actions into debugging episodes. As the program contained four bugs, we assigned the actions to four distinct debugging episodes. The assignment was informed by both the screen recordings and the accompanying audio data, which provided additional context on the students' intentions. We observed that the students worked on the bugs in an intersecting rather than strictly linear manner. Additionally, there was an initial exploratory phase in which the students interacted with the program without targeting a specific bug. Therefore, these actions were not assigned to any debugging episode.

Step 3: Mapping actions within each debugging episode to debugging process steps. In the third step, we mapped the actions for each debugging episode to the steps of the debugging process model by Araki et al. (1991). We chose this model as baseline, as it provides a fine-grained description of the ideal-typical debugging process and considers localizing and fixing the bug as two separate phases. To account for the fact that students were working with unfamiliar code, we extended the model with a code comprehension step. The mapping was conducted by the first author and validated by a second researcher who independently coded 25% of all data (one out of the four screen recordings). Inter-rater agreement was assessed using Cohen's kappa ($\kappa = 0.71$), indicating substantial agreement. Remaining discrepancies were discussed and resolved collaboratively.

We only considered actions for the mapping that were previously assigned to a debugging episode. Several actions could be directly mapped to a process step. For example, parsing the code with the mouse was mapped to the code comprehension step as the students seem to inspect the code in order to understand its structure. Especially for interpreting timeframes with no or random mouse movements, the audio data provided important information and thus indicated the debugging process step. For example, the utterance *"- Wait, what happens if we disable this part now?"* was mapped to hypothesis selection, as the students expressed an idea about a potential bug location they wanted to evaluate. However, not all actions could be mapped to a specific process step. Actions related to navigating the code (e.g., scrolling or changing the view) were therefore excluded from the mapping for this reason. In Table 1 we display the mapping for one exemplary debugging episode from the selected example.

Step 4: Comparing mapping with the ideal-typical debugging process. For each debugging episode the students' debugging processes were visualized according to the visualization of the debugging process model proposed by Araki et al. (1991). Refer to Figure 2b for two exemplary process visualizations. In particular, we analyzed whether process steps were omitted, repeated, or carried out in a different order, allowing us to identify deviations from the ideal-typical debugging process.

4.3 Insights from the Case Study

Across the four analyzed screen recordings (containing 19 debugging episodes), we observed some recurring patterns. These observations illustrate how the proposed approach reveals patterns in students' debugging processes that may indicate key sources of difficulty. Given the small and specific sample, these insights aim to illustrate detailed, context-specific aspects of students' debugging processes rather than to provide generalizable findings.

A frequently recurring pattern was that the students **did not verify their changes**. In some cases, fixes were not validated at all (refer to debugging episode B in Figure 2b), while in others, unsuccessful attempts were followed by further modifications without verifying whether the final solution actually resolved the bug (refer to debugging episode A in Figure 2b). While this behavior sometimes still resulted in a correct solution, in other cases the changes did not fix the bug or added new bugs to the program that remained unnoticed. As verification is essential for confirming correctness, its omission indicates that this step appears to be challenging for students. Beyond missing verification steps, the students also showed **trial-and-error approaches**. For example, the students modified the program despite explicitly stating that they do not understand the code (e.g., *"I don't know what that means, but let's just change it"* – debugging episode A in Table 1), suggesting that their program modification steps were not guided by explicit hypotheses about the program behavior. This suggests that their debugging was at times driven by trial-and-error rather than by reasoned hypotheses. Another observed pattern was **fragmented debugging processes** in

Observed behavior	Verbalization	Actions	Process-Step
While getting familiar with the program the students recognized that the tail sometimes does not follow the snake as it is supposed to be.	*"When you go down, it gets longer and longer. When you go up and across, it gets shorter and shorter."*	Execute the program	**Observe the failure**
	Break: Students are working on other bugs.		
While verifying a previous addressed bug, one student focuses again on the tail of the snake.	*"When it goes up and there..."*	—	**Observe the failure**
The students are discussing the code and are inspecting the different options behind the drop-down menus.	*"- What is [variable name]? [...] - It changes by three there, but not there. Oh, I think that's the length!"*	Inspect drop-down menu - Parsing Blocks	**Code Comprehension**
They decide to change values (from 3 to 0), without understanding the code properly.	*"I don't know what that means, but let's just change it."*	Inspect drop-down menu - Change value - Change value - Change value	**Program modification**
They check their changes by running the program. The changes were not successful. Now the snake gets longer with every step.	—	Execute the program	**Bug fixed? - No**
The students realize that they should change all the values (back) to 3. However, they still do not know why the fix is correct.	*"- We have to set [variable name] to three! - Yeah, I think so as well, because now its getting longer all the time. - I have no idea what [variable name] means."*	—	**Correction-hypothesis modification**
The students change the values (from 0 to 3).	*"Here. And there as well."*	Change back to initial value - Change value - Change back to initial value - Change value	**Program modification**
They run the program, but one student is already thinking out loud about another failure and so they are already discussing this problem and are not paying attention to the tail. This is not a proper verification whether the bug is already fixed.	*"- And just to check, right at the very beginning. Don't look down. - But right at the very beginning, what is she doing there?"*	Execute the program	—

Table 1: Mapping of exemplary actions to debugging process steps. For some actions, process steps were inferred from audio data. The corresponding debugging process is shown in Figure 2b, (debugging episode A), addressing the bug where the tail of the snake does not follow the head correctly.

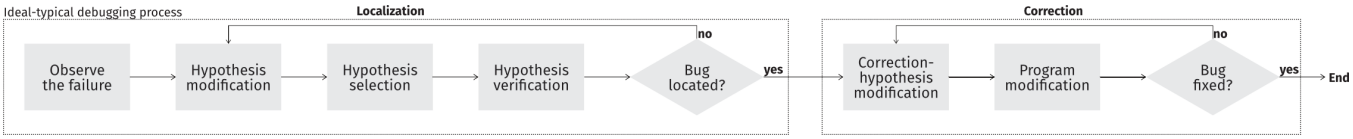
which students omitted several steps of the ideal-typical debugging process, resulting in incomplete or shortened debugging processes (refer to debugging episode A in Figure 2b). Nevertheless, these shortened processes often still resulted in a successful outcome, indicating that students' debugging processes can deviate from the ideal-typical process while still leading to correct solutions.

At the same time, the students also showcased debugging processes that corresponded to the ideal-typical debugging process model proposed by Araki et al. (1991). When fixes were unsuccessful, students often reflected on the outcome, undid changes, and revised their hypotheses, which aligns with several key elements of an ideal-typical debugging process.

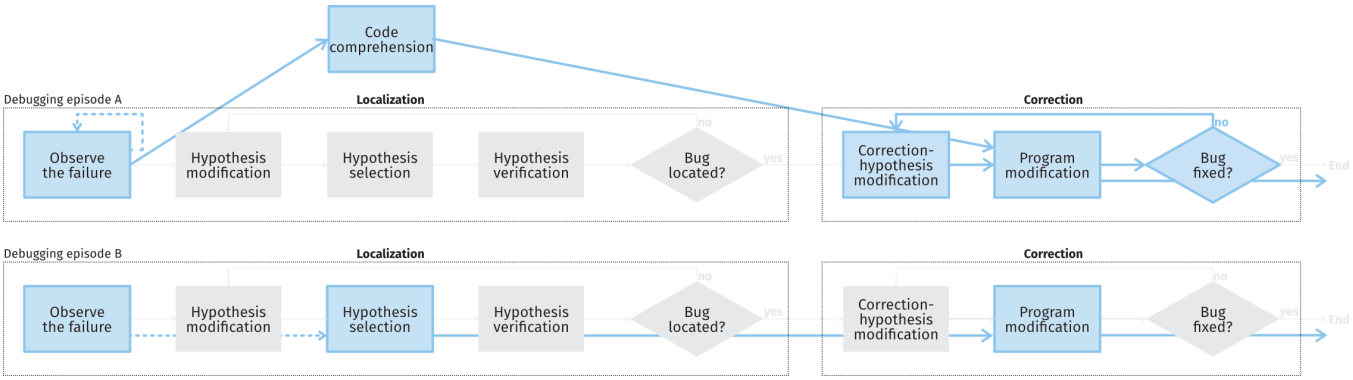
5 Discussion and Conclusion

This paper presents a theory-based approach to analyzing students' debugging processes and demonstrates its applicability and benefits through a case study. By applying this approach, we provide a programming language-independent perspective on students' debugging processes. The approach enables the systematic identification of deviations from an ideal-typical debugging process and reveals where and how students struggle during debugging. Such process-level information is difficult to capture using traditional outcome-based measures, underscoring the added value of process-oriented analyses.

A key methodological strength of the proposed approach lies in its **theory-based foundation**. In contrast to several prior studies that relied on inductive analyses [8, 23], the approach abstracts observable actions into debugging process steps grounded in an ideal-typical debugging process model. While prior research has primarily assessed intervention effects using pre–post performance measures [22], the approach enables the analysis of such effects at the process level by comparing students' debugging processes before and after an intervention. This makes it possible to examine not only whether interventions are effective, but how they change students' debugging processes. Furthermore, the approach extends the view on debugging processes by shifting from snapshot-based measures [1, 18] to a **process-oriented perspective**. Instead of relying on isolated program snapshots, it uses screen recordings that capture all actions performed by the students. This enables the reconstruction of entire debugging processes and reveals aspects that remain hidden in snapshot-based analyses. Beyond its theory-based and process-oriented foundation, the proposed approach connects identified process steps with the **sequential structure**



(a) Visualization of the debugging process model from Araki et al. (1991) – figure created and adapted by the authors.



(b) The process steps highlighted in blue were carried out by the students; the steps in gray were missing. Blue arrows mark transitions between steps. Dashed blue arrows indicate transitions that were interrupted by another debugging episode.

Figure 2: Visualization of debugging episodes, in contrast to the ideal-typical debugging process.

of the ideal-typical debugging process model. In doing so, it goes beyond the mere identification of individual process steps in previous approaches [16, 20]. This enables a structured comparison with an ideal-typical debugging process, allowing deviations to be systematically identified and providing a basis for analyzing potential sources of difficulty. An additional advantage of the approach is its **programming language-independence**, as the set of actions can be readily adapted to different programming languages. For example, the action *adding a block* in Scratch corresponds to *adding a line of code* in a text-based programming language, enabling transfer across programming languages.

The case study provides insights that highlight opportunities and offer initial perspectives for future investigations using the approach. Specifically, the analysis indicates that students' debugging processes often lack verification phases and that their changes are frequently based on trial-and-error rather than a solid understanding of the problem. Interestingly, some students were able to successfully debug their programs without following the ideal-typical debugging process, suggesting that successful outcomes do not necessarily imply the application of such a process. Overall, these observations underline the importance of considering both outcome and process, as outcome-based measures do not capture how students solve the problem.

While these insights highlight the value of a theory-based and process-oriented analysis, the high level of granularity makes the approach time-consuming and thus constitutes a key limitation. Consequently, the approach is currently best suited for in-depth analysis of small samples. At the same time, this fine-grained analysis provides a structured representation of debugging processes, which forms a foundation for future automated process analysis that opens up the possibility for large-scale analyses of debugging processes. It is important to note that the case study itself was based on a small and non-representative sample, which was intentionally used to illustrate the potential of the approach rather than to provide generalizable results.

To further develop the proposed approach, we aim to extend it by incorporating **multi-modal process data** such as eye-tracking, think-aloud protocols, and retrospective interviews. While these data sources have been used in prior research on debugging processes (e.g., [15, 17, 24]), they have typically been analyzed in isolation. Integrating these complementary data sources allows us to link observable actions with underlying cognitive processes, thereby providing a more comprehensive understanding of students' debugging processes. In particular, this triangulation supports the validation and refinement of the mapping of actions to debugging process steps.

In summary, by mapping students' observable actions to the steps of an ideal-typical debugging process, the proposed approach facilitates the reconstruction and investigation of entire debugging processes, rather than focusing on isolated strategies or performance outcomes. Its applicability was demonstrated in a case study, illustrating how the approach can be used to describe deviations from an ideal-typical debugging process in detail. Overall, this work contributes a methodological foundation for computing education research that aims to better understand learning processes and underlying mechanisms, rather than only focusing on learning outcomes.

References

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin* 37, 3 (2005), 84–88. https://doi.org/10.1145/1151954.1067472

[2] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. 1991. A general framework for debugging. *IEEE Software* 8, 3 (1991), 14–20. https://doi.org/10.1109/52.88939

[3] Renee C. Bryce, Alison Cooley, Amy Hansen, and Nare Hayrapetyan. 2010. A one year empirical study of student programming bugs. In *2010 IEEE Frontiers in Education Conference (FIE)*. IEEE, F1G–1–F1G–7. https://doi.org/10.1109/FIE.2010.5673143

[4] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. Identifying challenging CS1 concepts in a large problem dataset. In *Proceedings of the 45th ACM technical symposium on Computer science education* (New York, NY, USA), J. D. Dougherty, Kris Nagel, Adrienne Decker, and Kurt Eiselt (Eds.). ACM, 695–700. https://doi.org/10.1145/2538862.2538966

[5] David DeLiema, Jeffrey K. Bye, and Vijay Marupudi. 2024. Debugging Pathways: Open-Ended Discrepancy Noticing, Causal Reasoning, and Intervening. *ACM Transactions on Computing Education* 24, 2 (2024), 1–34. https://doi.org/10.1145/3650115

[6] David DeLiema, Maggie Dahn, Virginia J. Flood, Ana Asuncion, Dor Abrahamson, Noel Enyedy, and Francis Steen. 2020. Debugging as a Context for Fostering Reflection on Critical Thinking and Emotion. In *Deeper Learning, Dialogic Learning, and Critical Thinking*, Emmanuel Manalo (Ed.). Taylor & Francis Group.

[7] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference* (New York, NY, USA), Raina Mason and Simon (Eds.). ACM, 83–89. https://doi.org/10.1145/3160489.3160493

[8] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116. https://doi.org/10.1080/08993400802114508

[9] Laurie Gale and Sue Sentance. 2023. Investigating the Attitudes and Emotions of K-12 Students Towards Debugging. In *The United Kingdom and Ireland Computing Education Research (UKICER) conference* (New York, NY, USA), Troy Astarte, Faron Moller, Keith Quille, and Seán Russell (Eds.). ACM, 1–7. https://doi.org/10.1145/3610969.3611120

[10] David J. Gilmore. 1991. Models of debugging. *Acta Psychologica* 78, 1-3 (1991), 151–172. https://doi.org/10.1016/0001-6918(91)90009-O

[11] 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

[12] John D. Gould and Paul Drongowski. 1974. An Exploratory Study of Computer Program Debugging. *Human Factors: The Journal of the Human Factors and Ergonomics Society* 16, 3 (1974), 258–277. https://doi.org/10.1177/001872087401600308

[13] Gayithri Jayathirtha, Deborah Fields, and Yasmin Kafai. 2024. Distributed debugging with electronic textiles: understanding high school student pairs' problem-solving strategies, practices, and perspectives on repairing physical computing projects. *Computer Science Education* 34, 4 (2024), 718–752. https://doi.org/10.1080/08993408.2023.2297738

[14] Irvin R. Katz and John R. Anderson. 1987. Debugging: an analysis of bug-location strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399. https://doi.org/10.1207/s15327051hci0304_2

[15] ChanMin Kim, Emre Dinç, Eunseo Lee, Afaf Baabdullah, Anna Y. Zhang, and Brian R. Belland. 2023. Revisiting Analogical Reasoning in Computing Education: Use of Similarities Between Robot Programming Tasks in Debugging. *Journal of Educational Computing Research* 61, 5 (2023), 1036–1063. https://doi.org/10.1177/07356331221142912

[16] ChanMin Kim, Jiangmei Yuan, Lucas Vasconcelos, Minyoung Shin, and Roger B. Hill. 2018. Debugging during block-based programming. *Instructional Science* 46, 5 (2018), 767–787. https://doi.org/10.1007/s11251-018-9453-5

[17] Yu-Tzu Lin, Cheng-Chih Wu, Ting-Yun Hou, Yu-Chih Lin, Fang-Ying Yang, and Chia-Hu Chang. 2016. Tracking Students' Cognitive Processes During Program Debugging—An Eye-Movement Approach. *IEEE Transactions on Education* 59, 3 (2016), 175–186. https://doi.org/10.1109/TE.2015.2487341

[18] Rifat Sabbir Mansur, Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. 2020. Exploring the Bug Investigation Techniques of Intermediate Student Programmers. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (New York, NY, USA) *(Koli Calling '20)*. Association for Computing Machinery. https://doi.org/10.1145/3428029.3428040

[19] Tilman Michaeli and Ralf Romeike. 2019. Improving Debugging Skills in the Classroom - The Effects of Teaching a Systematic Debugging Process. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education* (New York, NY, USA). ACM, 1–7. https://doi.org/10.1145/3361721.3361724

[20] Meghan M. Parkinson, Seppe Hermans, David Gijbels, and Daniel L. Dinsmore. 2024. Exploring debugging processes and regulation strategies during collaborative coding tasks among elementary and secondary students. *Computer Science Education* 34, 4 (2024), 617–644. https://doi.org/10.1080/08993408.2024.2305026

[21] David N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

[22] Elena Spörer and Tilman Michaeli. 2025. Investigating Debugging Processes: A Scoping Review. In *Proceedings of the 25th Koli Calling International Conference on Computing Education Research* (New York, NY, USA), Juho Leinonen and Rodrigo Duran (Eds.). ACM, 1–12. https://doi.org/10.1145/3769994.3770005

[23] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Analysis of a Process for Introductory Debugging. In *Proceedings of the 23rd Australasian Computing Education Conference* (New York, NY, USA), Claudia Szabo and Judy Sheard (Eds.). ACM, 11–20. https://doi.org/10.1145/3441636.3442300

[24] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2023. A Think-Aloud Study of Novice Debugging. *ACM Transactions on Computing Education* 23, 2 (2023), 1–38. https://doi.org/10.1145/3589004

[25] Wei Yan, Feiya Luo, Maya Israel, Ruohan Liu, and Latoya T. Chandler. 2025. How Do Elementary Students Apply Debugging Strategies in a Block-Based Programming Environment? *Education Sciences* 15, 3 (2025), 292. https://doi.org/10.3390/educsci15030292

[26] Stephanie Yang, Miles Baird, Eleanor O'Rourke, Karen Brennan, and Bertrand Schneider. 2024. Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions. *ACM Transactions on Computing Education* (2024). https://doi.org/10.1145/3690652

[27] Byung-Do Yoon and O. N. Garcia. 1998. Cognitive activities and support in debugging. In *Proceedings Fourth Annual Symposium on Human Interaction with Complex Systems*. IEEE Comput. Soc, 160–169. https://doi.org/10.1109/HUICS.1998.659974

[28] Andreas Zeller. 2009. *Why programs fail*. Morgan Kaufmann an imprint of Elsevier and dpunkt.verlag.