Baltic J. Modern Computing, Vol. 13 (2025), No. 3, 641-655
https://doi.org/10.22364/bjmc.2025.13.3.05

Constructive-synthesizing Modeling of Debugging Process Visualization

Viktor SHYNKARENKO, Oleksandr ZHEVAHO

Department of Computer Information Technology, Ukrainian State University of Science and Technologies, Lazariana 2, Dnipro, 49010, Ukraine

shinkarenko_vi@ua.fm, o.o.zhevaho@ust.edu.ua

ORCID 0000-0001-8738-7225, ORCID 0000-0003-0019-8320

Abstract: This paper presents a theoretical framework for developing video-based debugging instruction using constructive-synthesizing modeling (CSM). The proposed framework formalizes the transformation of integrated development environment debugging logs into educational videos through CSM methodology, providing a systematic approach for creating visual representations of debugging processes. The CSM formalization enables reproducible conversion of temporal debugging sequences into structured educational content, with videos featuring synchronized code context, debugging actions, and explanatory annotations. We present a seven-phase implementation methodology for classroom integration, including structured video analysis, collaborative reflection, and expert-guided discussion phases. The framework establishes theoretical foundations for future empirical validation of video-based debugging instruction, addressing the critical gap between debugging's importance in professional practice and its treatment in educational contexts.

Keywords: debugging, constructive-synthesizing modeling, visualization, software engineering, education, information technology

1. Introduction

In modern programming curricula, the primary focus is on writing code, while tasks related to debugging are often relegated to a secondary position (Michaeli and Romeike, 2019a; Rich et al., 2019). This creates a misconception that debugging is less crucial than programming. However, in real-world, these two processes are inextricably linked and require equal attention.

Although debugging is a critically important component of programming, accounting for an estimated 40 to 60% of the time spent on programming, learning debugging strategies is often a skill that beginners are expected to develop on their own (Alaboudi and Latoza, 2023). As a result, students primarily learn debugging by working through their own mistakes, which can be quite frustrating. They must spend considerable time and effort hypothesizing the causes of errors while simultaneously tackling other cognitively demanding tasks, such as understanding and writing code (Ma et al., 2024).

The purpose of this research is to develop a formal theoretical framework for converting debugging logs into educational videos using CSM, and to propose a classroom

methodology based on this visualization, establishing a foundation for future empirical studies on video-based debugging education.

This paper addresses the gap between debugging importance and debugging instruction through a formal approach to debugging visualization. We explicitly note that this paper focuses on the theoretical framework. We provide the formal model for debugging process visualization, ensuring systematic and reproducible video generation. Empirical validation of educational effectiveness is beyond the current scope and represents important future work.

2. Related Work

2.1. The Educational Challenge: Debugging Instruction Gap

Despite debugging's critical importance in professional software development, it remains severely underrepresented in formal computer science curricula. Recent systematic reviews confirm that debugging instruction is an underexplored area, with most educational interventions focusing on programming syntax rather than error detection and resolution strategies (Yang et al., 2024). The complexity of debugging as a cognitive process-requiring hypothesis formation, systematic testing, and analytical reasoning-necessitates specialized pedagogical approaches distinct from traditional programming instruction (Jemmali et al., 2020).

Current educational practices typically relegate debugging to incidental learning during programming assignments rather than explicit instruction (O'Dell, 2017). This approach fails to prepare students adequately, as evidenced by surveys showing that professional programmers receive virtually no formal debugging training, with most learning through self-directed workplace experience (Whalley et al., 2021). This educational gap creates a paradox where one of the most time-consuming aspects of professional development receives minimal systematic attention in academic settings.

2.2. Evidence for Systematic Debugging Instruction and Pedagogical Approaches

Empirical evidence demonstrates that debugging skills can be effectively taught through structured approaches. Systematic reviews of debugging interventions from 2010-2022 reveal that well-designed programs consistently improve students' debugging accuracy, efficiency, and self-efficacy (Yang et al., 2024). Research has identified three critical cognitive processes in debugging: identification, isolation, and iteration (Hylton et al., 2023), providing concrete frameworks for instructional design.

Effective pedagogical approaches range from traditional hands-on programming (Michaeli and Romeike, 2019a) to structured error introduction methods (Kerslake, 2024). The scientific method has emerged as an influential framework, treating debugging as hypothesis-driven investigation through five sequential stages (O'Dell, 2017). Multi-language approaches demonstrate effectiveness in developing generalizable skills (Wilkin, 2025), while contemporary research emphasizes combining direct instruction with deliberate practice (Yamamoto et al., 2016).

However, studies reveal that novice and expert debugging strategies differ substantially, with beginners struggling to generate comprehensive hypotheses and

lacking systematic evaluation strategies (Caballero et al., 2021; Liu and Paquette, 2023; Ma et al., 2024). Students typically exhibit surface-level approaches, prioritizing immediate fixes over systematic investigation (Hylton et al., 2023). These cognitive differences necessitate personalized learning approaches that address varying strengths across different knowledge domains (Ding et al., 2024; Zhang et al., 2022). Notably, even experienced educators show substantial variance in diagnostic approaches to identical scenarios (Wachter and Michaeli, 2024).

2.3. Technology-Enhanced Instruction and Observational Challenges

Technological advances have expanded debugging instruction possibilities through AI-powered environments, visualization tools, and automated testing frameworks. Large language models enable novel approaches like the HypoCompass system, where students act as teaching assistants correcting LLM-generated code (Ma et al., 2024). Program visualization tools provide dynamic representations of execution flow, while automated testing frameworks offer systematic error detection approaches (Caballero et al., 2021). Enhanced error messaging systems show potential for improving debugging performance through better feedback mechanisms (Kerslake, 2024).

However, a fundamental challenge exists: most student debugging occurs outside classroom settings, limiting educators' ability to observe and support skill development. Students debug autonomously during homework and personal projects, creating gaps between instruction and independent practice (Liu and Paquette, 2023). Researchers have explored various approaches for capturing debugging behaviors, including IDE debugging logs (Shynkarenko and Zhevaho, 2020a) and think-aloud protocols, but these methods face scalability limitations.

2.4. Video-Based Approaches and Theoretical Framework Requirements

Video-based instruction represents an emerging frontier for addressing debugging education challenges. Research demonstrates effectiveness of video-based learning for conveying complex procedural knowledge in programming contexts. Video vignettes prove particularly valuable for teacher education, enabling repeated observation of debugging scenarios difficult to capture naturally (Wachter and Michaeli, 2025; Ding et al., 2024; Hylton et al., 2023). These approaches offer dynamic representation of expert debugging processes, including temporal sequences of decisions and adaptive strategies.

The application of CSM to debugging instruction represents a novel theoretical contribution. While CSM has proven effective in various educational contexts for transforming complex processes into structured learning materials, its systematic application to debugging visualization remains largely unexplored. The systematic transformation of IDE debugging logs into educational videos through CSM methodology addresses reproducibility challenges that have limited the scalability and theoretical grounding of previous debugging instruction research.

This convergence of video-based instruction, debugging education requirements, and formal modeling approaches establishes the foundation for systematic, theoretically-grounded instructional designs. The integration addresses multiple challenges simultaneously: providing observable representations of expert debugging processes, creating reusable instructional materials extending beyond individual classroom contexts, and establishing systematic frameworks for debugging instruction that can be empirically

evaluated and continuously improved. These foundations directly motivate the need for a formal theoretical framework that can systematically transform debugging processes into effective educational visualizations.

3. Theoretical Framework: Constructive-Synthesizing Modeling for Debugging Visualization

This section presents the application of CSM to debugging visualization, establishing a framework for converting debugging logs into educational videos. CSM provides a formal methodology for representing complex processes and their transformations by linking elements while considering their attributes, aggregates, and relational structures. The fundamental principles and ontological foundations of CSM, including the generalized constructor framework, have been established in previous work (Skalozub et al., 2017). Examples of CSM applications demonstrate its versatility across various domains (Shynkarenko and Zhevaho, 2020a; Shynkarenko and Zhevaho, 2021).

CSM enables formal description and transformation of complex process data. In our system, IDE-generated logs are modeled as structured data and transformed into visual educational content. The CSM approach is suitable because:

- it allows hierarchical decomposition of debugging events;
- it facilitates rule-based transformations and interpretations;
- it supports reproducibility and systematization in educational content generation.

We detail the formalization of this process through constructors, terminals, and interpretation rules. Each debugging session is treated as a sequence of events, where frames are generated per event and assembled chronologically into a video.

Previous applications of CSM to debugging contexts formalized the data collection process, resulting in a constructor for generating debugging action logs and a Microsoft Visual Studio IDE extension that records debugging activities in structured event logs (Shynkarenko and Zhevaho, 2020b).

The current research extends this foundation by formalizing the conversion of textual debugging logs into video-based visualizations. This transformation addresses the observational challenges identified in debugging education while providing a theoretically grounded approach to creating scalable educational content that captures the temporal, contextual, and strategic dimensions of expert debugging practices.

The first stage of this construction involves specializing the generalized constructor

$$C = \langle M, \Sigma, \Lambda \rangle \ {}_{S}\mapsto\ C_V = \langle M_V, \Sigma_V, \Lambda_V \rangle,$$

where C – generalized constructor; M – carrier, which includes terminals and non-terminals, as well as a set of rules; Σ – a set of operations and relationships for elements M; Λ – information support for construction; ${}_{S}\mapsto$ – specialization operation of the constructor.

Terminals and their attributes:

- ${}_{sessions}log$ – debugging event log, which contains an array of debugging sessions;
- ${}_{index,size}sessions$ – array of sessions with its attributes: *index* – the index of the session in the array, *size* – the size of the array;

- $_{events,start,end}$*session* – a debugging session, which consists of an events array and the timestamps for the *start* and *end* of the debugging process;
- $_{index,size}$*events* – array of events with its attributes: *index* – the index of the *event* in the array, *size* – the size of the array;
- $_{name,context,\ timestamp}$*event* – an event in the development environment during debugging with its attributes: *name* – the name of the event, *context* – the context, a dynamic structure object containing information about the event's surroundings, such as the file the user is working with, the line number where the event occurred, etc. (the structure of the object may vary for different events), *timestamp* – timestamp when the event occurred;
- $_{frames}$*video* – a generated video file consisting of a frames sequence;
- $_{index,size}$*frames* – array of frames with its attributes: *index* – the index of the *frame* in the array, *size* – the size of the array;
- $_{info,code,comment}$*frame* – a generated frame for the video, based on an event from the IDE with its attributes: *info* – the upper section of the frame, containing information about the debugging (number of sessions, total session time), *code* – the main section of the frame, which includes code and visual elements of the event, *comment* – the side section, containing a natural language explanation of the event.

Operations on attributes:

- ○*(t)* – operation of setting the attribute values of terminal *t* by an external executor;
- ≈*(a,i,t)* – operation of retrieving the element at index *i* from array *a* and setting its value in terminal *t*;
- ∴*(a,i,f,b)* – operation of generating frame *f* based on the element retrieved at index *i* from array *a* and adding it to array *b* at index *i;*
- ≡*(lo,ro,t)* – operation of comparing the value *lo* with the value *ro*, and setting 1 if they are equal, or 0 otherwise, in *t*;
- ≠*(lo,ro,t)* – operation of comparing the value *lo* with the value *ro*, and setting 1 if they are **not** equal, or 0 otherwise, in *t*;
- ±*(video)* – operation of creating a video based on the frames from the given terminal *video*.

Let's perform the interpretation

$$\langle C_V = \langle M_V, \Sigma_V, \Lambda_V\rangle, C_A = \langle M_A, V_A, \Sigma_A, \Lambda_A\rangle\rangle \ {}_{I}\mapsto \langle C_{VAI} = \langle M_{VAI}, \Sigma_{VAI}, \Lambda_{VAI}\rangle\rangle,$$

where V_A – set of algorithms for forming in the basic algorithmic structure (BAS); ${}_{I}\mapsto$ – interpretation operation.

C_{VAI} in addition to the algorithms for executing operations from the BAS, it includes the following algorithms: $A_1|_t^t$, $A_2|_{a,i,t}^{\ t}$, $A_3|_{a,i,f,b}^{\ b}$, $A_4|_{lo,ro,t}^{\ t}$, $A_5|_{lo,ro,t}^{\ t}$, $A_6|_{video}^{video}$, which perform the corresponding operations: ○*(t)*, ≈*(a,i,t)*, ∴*(a,i,f,b)*, ≡*(lo,ro,t)*, ≠*(lo,ro,t)*, ±*(video)*.

Let's perform the concretization

$$C_{VAI} \mapsto_K C_{VAIK}\ (\text{LOG}) = \langle M_K, \Sigma_K, \Lambda_K\rangle,$$

where LOG – event log provided for conversion into a video; $\mapsto_K$ – concretization operation.

Construction initial condition: σ – the non-terminal from which the derivation begins, with initial values $i = 1$, $j = 1$, and also $t_0 = 1$, $t_1 = 0$. If $t_i = 0$, the corresponding rule is not executed.

Construction completion condition: all sessions and events from the external executor's event log are processed and a video is generated.

Sequential execution of rules will be denoted as $\prod_{i=1}^{n} s_i$.

The first rule performs the initial processing of the provided event log

$$s_1 =< \sigma \rightarrow \log\bullet \prod_{i=1}^{size\,\downarrow sessions\,\downarrow log} \alpha_i >,$$

$$g_1 =<\circ\,(\log) >,$$

where $b \downarrow a$ – identifies attribute b of object a.

Each log consists of debugging sessions, and each session consists of events series. An event refers to actions such as setting a breakpoint, step-by-step execution of code, viewing the call stack, and other standard debugging tools in the IDE, each accompanied by relevant contextual information, such as the file name and line number.

The second rule involves processing the sessions

$$s_2 =< \alpha_i \rightarrow \text{session}\bullet \prod_{j=1}^{size\,\downarrow events\,\downarrow session} \beta_j >,$$

$$g_2 =<\approx\,(sessions \downarrow \log, \text{i}, \text{session}) >.$$

Each event has an action and context. The action determines the visual effect and comment to be applied in the frame. The context specifies the particular file and line number to be displayed in the main area of the frame.

The third rule involves creating a frame for each event

$$s_3 =< \beta_{j\ t_0} \rightarrow \text{frame}\bullet\ \beta_j >,$$

$$\begin{gathered} g_3 =<\vdots\,(events \downarrow \text{session}, \text{j}, \text{frame}, \text{frames} \downarrow \text{video}), \\ \equiv (j, size \downarrow events \downarrow session, t_1), \\ \neq (j, size \downarrow events \downarrow session, t_0) >. \end{gathered}$$

The video consists of individual frames, where each frame visually represents a separate event or state transition in the debugging process (Figure 1).

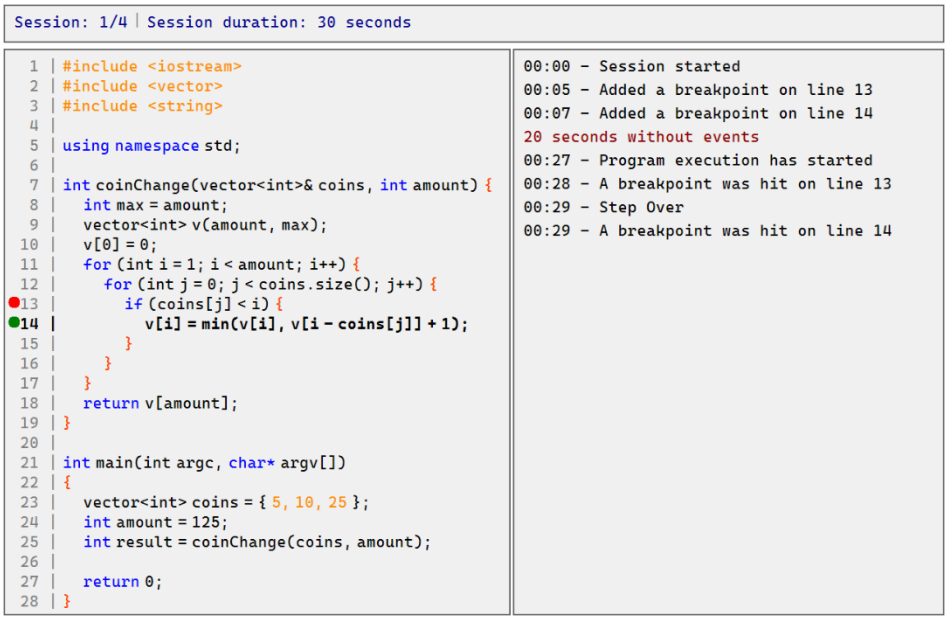

Figure 1. Example frame from the debugging process visualization

Each frame is composed of three areas (Shynkarenko and Zhevaho, 2024):

- upper area (process information): displays aggregated data from session logs, such as the number of sessions and their duration. This area remains unchanged for all frames of the current session;
- main area (program text): the context (file name and line number) of each event is used to display the corresponding section of the code. This area is dynamically updated as the video progresses through the events;
- side area (event comments): each event is marked with a timestamp and accompanied by a brief explanation, such as "Breakpoint set at line 42." If there is a pause of more than 5 seconds between events, a comment indicating the duration of inactivity is inserted.

Actions like setting or hitting a breakpoint trigger a visual cue, highlighting the corresponding line in the main area.

The fourth rule involves the final assembly of frames into a single video file after all events have been processed

$$s_4 =< \beta_{j\ t_1} \rightarrow video >,$$

$$g_4 =< \pm(\text{video}) >.$$

Frames are arranged in chronological order based on their timestamps, preserving the sequence of events that occurred during the debugging session. Periods of inactivity are marked with comments in the appropriate area, and the video transitions to the next active frame to ensure that the video is as informative as possible, without an excessive number of static frames where nothing happens.

4. Proposed Teaching Methodology

This section outlines an approach to teaching debugging programs based on process visualization videos (Shynkarenko and Zhevaho, 2024). The implementation of the CSM-based debugging visualization follows a structured seven-phase approach designed to maximize learning effectiveness while providing measurable outcomes for future empirical validation. The process consists of the following key steps (Figure 2):

4.1. Task assignment

Students receive programming assignments or debugging challenges carefully selected to demonstrate specific debugging strategies or common error patterns. The task selection process emphasizes bugs that represent typical debugging scenarios encountered in professional practice, ensuring that students engage with realistic problem-solving contexts. Clear task objectives and expected outcomes are provided to establish learning goals, while bug complexity is calibrated to match students' current skill levels to maintain appropriate challenge without overwhelming novice debuggers. The assignments incorporate both syntax and logic errors to provide comprehensive practice across different categories of debugging challenges, enabling students to develop versatile error detection and resolution capabilities.

4.2. Action logging

Students work on the debugging task while specialized IDE extensions automatically capture all debugging actions during their work sessions (Shynkarenko and Zhevaho, 2020b). The logging system records timestamps, action types, code contexts, and decision points without interfering with the debugging process. Successful implementation requires ensuring that debugging extensions are properly installed and activated across all student workstations prior to session commencement. Instructors must verify that log collection mechanisms are functioning correctly before students begin their debugging work and establish appropriate time limits for debugging sessions to maintain focused learning experiences. Students are informed about data collection procedures and privacy measures to maintain transparency and comply with educational data protection requirements, while the non-intrusive nature of the logging system allows students to engage naturally with debugging tasks without awareness of the monitoring process affecting their problem-solving approaches.

4.3. Video creation

Based on the CSM framework, the constructor processes collected logs to generate standardized educational videos from representative debugging sessions selected for their pedagogical value (Shynkarenko and Zhevaho, 2024). Each video maintains consistent visual formatting with session metadata displayed in the upper area, code context in the main viewing area, and action annotations positioned in the side area, incorporating synchronized code visualization, timing information, and strategically placed highlights. The automated generation process allows for configuration of video parameters including playback speed, annotation density, and visual emphasis to optimize educational effectiveness. Generated videos undergo quality review to ensure educational value and

clarity before classroom implementation, with supplementary materials prepared as needed to support specific learning objectives and provide additional context for complex debugging scenarios.

4.4. Video presentation

The instructional video is shown in class, serving as a foundation for discussion and analysis. Students view generated videos in classroom settings through structured observation protocols that begin with introducing the debugging scenario and learning objectives to establish context. Video playback incorporates strategic pauses at critical junctures to facilitate discussion and allow students to process complex debugging decisions. During viewing, students are encouraged to take detailed notes on debugging strategies, patterns, and decision points, while initial observation focuses on identifying debugging approaches without instructor commentary to promote independent analysis. The instructor highlights key decision points and debugging techniques during subsequent discussion phases, and multiple videos can be compared to demonstrate different debugging approaches to the same or similar problems, providing students with comprehensive exposure to various problem-solving methodologies and enabling comparative analysis of debugging effectiveness.

4.5. Initial analysis

Students provide initial observations and analysis of the debugging process shown in the video, developing critical thinking and analytical skills through structured examination of debugging strategies. Initial analysis involves students identifying and commenting on key actions visible in the video, such as incorrect breakpoints or inefficient navigation, while focusing on strategy identification to determine what debugging approaches were employed. Students examine decision points to understand why certain actions were taken, evaluate the effectiveness of strategies to identify which techniques worked well, and consider alternatives to explore what could have been done differently. This phase emphasizes hypothesis generation and evaluation as students observe debugging strategies, identify effective techniques, and propose potential improvements. The teacher assesses students' understanding through their analytical commentary and outlines areas for improvement, fostering metacognitive awareness as students develop deeper insights into debugging methodology and decision-making processes.

4.6. Feedback collection

Structured collection of observations from both students and instructor using standardized forms ensures comprehensive documentation of the debugging process analysis. Students document their observations using standardized forms that capture technical observations about specific debugging techniques used, process analysis of the overall debugging workflow, learning insights regarding new strategies discovered, and improvement suggestions for better approaches identified. These forms systematically cover identified debugging strategies, decision rationale, alternative approaches, and error patterns, creating a foundation for comparative analysis across different debugging sessions. Teacher assessments complement student perspectives by providing expert evaluation of strategy effectiveness and learning outcomes, while the structured feedback format

enables systematic data collection that supports both immediate learning objectives and longer-term curriculum refinement based on observed student comprehension patterns and skill development trajectories.

4.7. Discussion

Facilitated comprehensive discussion consolidates learning and develops debugging best practices as students and teacher exchange opinions and analyze the debugging process. This phase enables identification of misconceptions and reinforcement of effective practices through comparative analysis of different debugging strategies observed during the video session. Students work collaboratively to identify common debugging patterns and anti-patterns, developing a shared understanding of what constitutes effective versus problematic debugging approaches. The discussion synthesizes observations and insights to help students internalize effective debugging strategies while developing debugging heuristics for future application in their own programming work. Through guided conversation, students develop metacognitive awareness of their debugging processes and create personal debugging strategy guides that capture the most valuable techniques and decision-making frameworks learned during the session, establishing individualized references that can support their ongoing debugging skill development and promote transfer of learning to new programming contexts.

Figure 2. Steps of teaching debugging methodology

After the first discussion, the video is shown again, giving students the opportunity to review their conclusions and further analyze the programmer's actions. The final

discussion helps to consolidate skills and deepen understanding of the debugging process. This iterative approach allows observing how students' understanding of the debugging process has changed after the discussion and exchange of opinions.

This methodology enables students to improve their debugging skills while developing critical thinking. The iterative approach, realized through repeated viewing and analysis, allows for a deeper understanding of the debugging process. Comparative analysis of students' opinions with the teacher's expert assessment helps identify gaps in understanding and adjust the learning process accordingly. Additionally, by recording all student activities in logs, which are then used to generate videos, this approach also helps prevent cheating, as it becomes clear when tasks were not completed independently.

This approach addresses three critical limitations in current debugging education: (1) lack of process observability, (2) insufficient authentic examples, and (3) limited opportunities for comparative analysis. The video-based method enables students to observe debugging processes that would otherwise remain invisible during independent work.

5. Discussion

5.1. Theoretical Contributions

This research establishes three primary theoretical contributions to debugging education. First, it provides the formal framework for debugging process visualization using CSM methodology, enabling systematic analysis and comparison of debugging strategies across different educational contexts. Second, it defines systematic transformation rules for converting temporal debugging events into structured educational content, creating reproducible pathways from raw debugging data to pedagogical materials. Third, it establishes a theoretical foundation for video-based debugging instruction that can guide future empirical research and curriculum development.

The CSM-based approach addresses a fundamental challenge in debugging education: converting invisible cognitive processes into observable educational content. By formalizing the transformation of debugging logs into educational videos, the framework enhances process visibility while maintaining rigor and practical applicability. The automated video generation methodology enables systematic debugging education without requiring extensive instructor expertise in video production, offering a scalable solution to the widespread gap between debugging's professional importance and its educational treatment.

The formal framework structure enables systematic replication and adaptation across different educational environments and debugging platforms. This standardization potential represents a significant advancement over ad hoc approaches that have limited transferability and theoretical grounding. The formal foundation provided by CSM ensures that the video generation process can be systematically evaluated, refined, and extended to accommodate diverse educational contexts and student populations.

5.2. Implementation Considerations

While the theoretical framework demonstrates significant potential, several practical considerations must be addressed for successful implementation. Scalability represents a primary concern, as large-scale deployment requires efficient algorithms for log

processing and video generation. The computational complexity increases with session length and event frequency, necessitating optimization strategies to ensure practical deployment in resource-constrained educational environments.

The framework's current focus on IDE-based debugging activities may limit universal applicability across diverse development environments. Adaptation to command-line debugging, web development platforms, or mobile development environments may require framework extensions to accommodate different debugging tools and practices. However, the formal CSM foundation provides a systematic basis for such extensions, suggesting that adaptability challenges represent implementation rather than fundamental theoretical limitations.

Privacy and ethical considerations require careful attention throughout implementation. Comprehensive activity logging raises legitimate privacy concerns that must be addressed through transparent data collection policies, secure storage practices, and explicit user consent protocols. Students must be clearly informed about what data is collected, how it will be used, who can access it, and how long it will be retained. Clear opt-out mechanisms should be provided for non-educational activities, ensuring that personal projects remain private while maintaining the educational benefits of the system.

5.3. Framework Validation

The theoretical nature of this contribution necessitates comprehensive empirical validation to demonstrate practical effectiveness. Future validation should employ mixed-methods approaches combining quantitative performance measures with qualitative learning outcome assessments. Quantitative measures should include pre/post debugging task completion times, error identification accuracy rates, debugging strategy sophistication scores, and long-term retention assessments to establish measurable learning outcomes.

Qualitative validation requires student confidence self-reports, debugging strategy articulation quality assessments, instructor observation protocols, and focus group feedback on video effectiveness. Think-aloud protocols during video viewing can provide insights into cognitive processing, while longitudinal tracking of debugging competency development can assess sustained learning impacts.

Controlled experiments comparing video-based instruction with traditional debugging education approaches will establish relative effectiveness across diverse student populations. These studies should include randomized controlled trials with pre/post assessments and qualitative investigations of student learning experiences to provide comprehensive evaluation of the framework's educational value.

5.4. Future Research Directions

Several critical research directions emerge from this theoretical foundation. Empirical validation studies represent the immediate priority, requiring controlled experiments to demonstrate the framework's educational effectiveness compared to traditional approaches. Cognitive load assessment investigations will determine how video complexity and presentation timing affect learning outcomes for students with varying programming experience levels.

Cross-platform adaptation research can extend the framework to support multiple IDEs and programming languages, enhancing universal applicability and addressing current limitations in platform coverage. Long-term retention studies will provide longitudinal

assessment of debugging skill retention following video-based instruction, establishing the durability of learning outcomes.

The development of adaptive video generation capabilities represents an advanced research direction, enabling personalized video content based on individual learning patterns and debugging skill levels. Such adaptations could optimize learning efficiency while accommodating diverse student needs and learning preferences.

Beyond debugging education, the formal CSM approach provides a foundation for process visualization across multiple software engineering domains. Future research could extend the framework to support code review, testing, and refactoring education, establishing a comprehensive approach to software engineering process visualization that addresses broader educational challenges in computer science curriculum.

6. Conclusion

This paper presents a comprehensive theoretical framework for debugging education through video visualization, formalized using CSM methodology. The research addresses a critical gap in computer science education by providing the approach to transforming debugging processes into structured educational content. While empirical validation remains essential for demonstrating practical effectiveness, this work establishes a theoretical foundation that enables rigorous conceptualization of debugging visualization and provides a blueprint for future implementation and testing.

The primary contribution lies in the formal specification of systematic transformation rules for converting IDE debugging logs into educational videos with synchronized content across three complementary information areas. This CSM-based formalization ensures consistent, reproducible generation of educational materials from authentic debugging sessions. The framework defines terminal specifications and operational procedures that enable educators to systematically create video-based instruction without requiring extensive technical expertise in video production.

The seven-phase teaching methodology provides a structured implementation pathway that bridges theoretical framework with classroom practice. This methodology offers educators concrete guidance for integrating video-based debugging instruction while maintaining the authenticity and complexity of real-world debugging scenarios.

Future research will focus on empirical validation through controlled classroom studies and comparative effectiveness assessments to demonstrate the framework's practical value. The formal CSM foundation provides a robust basis for framework extension to additional software engineering processes, suggesting broader applications in computer science education. The theoretical groundwork established here enables systematic investigation of video-based debugging instruction effectiveness, supporting the development of evidence-based debugging education practices that can ultimately improve student preparation for professional software development challenges.

References

Alaboudi, A., Latoza, T. D. (2023). What constitutes debugging? An exploratory study of debugging episodes, *Empirical Software Engineering*, 28(5), 117. doi: 10.1007/s10664-023-10352-5.

Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S. (2021). A unified framework for declarative debugging and testing, *Information and Software Technology*, 129, 106427. doi: 10.1016/j.infsof.2020.106427.

Ding, L., Stacey, K. J., Yoon, M. (2024). Dialogue alongside or within lecturing videos for teaching debugging, *Journal of Research on Technology in Education*, pp. 1–18. doi: 10.1080/15391523.2024.2404133.

Hylton, D., Sung, S. H., Ding, X., Van Vleet, M. J. (2023). Board 196: A Framework to Assess Debugging Skills for Computational Thinking in Science and Engineering, *2023 ASEE Annual Conference and Exposition - The Harbor of Engineering: Education for 130 Years*. doi: 10.18260/1-2--42591.

Jemmali, C., Kleinman, E., Bunian, S., Almeda, M. V., Rowe, E., Seif El-Nasr, M. (2020). MAADS: Mixed-methods approach for the analysis of debugging sequences of beginner programmers, *51st ACM Technical Symposium on Computer Science Education*, pp. 86–92. doi: 10.1145/3328778.3366824.

Kerslake, C. (2024). Stump-the-Teacher: Using Student-generated Examples during Explicit Debugging Instruction, *55th ACM Technical Symposium on Computer Science Education*, pp. 653–658. doi: 10.1145/3626252.3630934.

Liu, Q., Paquette, L. (2023). Using submission log data to investigate novice programmers' employment of debugging strategies, *13th International Learning Analytics and Knowledge Conference*, pp. 637–643. doi: 10.1145/3576050.3576094.

Ma, Q., Shen, H., Koedinger, K., Wu, S. T. (2024). How to Teach Programming in the AI Era? Using LLMs as a Teachable Agent for Debugging, *Artificial Intelligence in Education, AIED 2024, Lecture Notes in Computer Science*, 14829, pp. 265–279. doi: 10.1007/978-3-031-64302-6_19.

Michaeli, T., Romeike, R. (2019a). Current status and perspectives of debugging in the K12 classroom: a qualitative study, *2019 IEEE Global Engineering Education Conference*, pp. 1030–1038. doi: 10.1109/educon.2019.8725282.

Michaeli, T., Romeike, R. (2019b). Improving Debugging Skills in the Classroom, *14th Workshop in Primary and Secondary Computing Education*, 15, pp. 1–7. doi: 10.1145/3361721.3361724.

O'Dell, D. H. (2017). The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills, *Queue*, 15(1), pp. 71–90. doi: 10.1145/3055301.3068754.

Rich, K. M., Strickland, C., Binkowski, T. A., Franklin, D. (2019). A K-8 Debugging Learning Trajectory Derived from Research Literature, *50th ACM Technical Symposium on Computer Science Education*, pp. 745–751. doi: 10.1145/3287324.3287396.

Shynkarenko, V., Zhevaho, O. (2020a). Constructive Modeling of the Software Development Process for Modern Code Review, *2020 IEEE 15th International Conference on Computer Sciences and Information Technologies (CSIT)*, pp. 392–395. doi: 10.1109/csit49958.2020.9322002.

Shynkarenko, V., Zhevaho, O. (2020b). Development of a toolkit for analyzing software debugging processes using the constructive approach, *Eastern-European Journal of Enterprise Technologies*, 5(2 (107)), pp.29–38. doi: 10.15587/1729-4061.2020.215090.

Shynkarenko, V., Zhevaho, O. (2021). Application of Constructive Modeling and Process Mining Approaches to the Study of Source Code Development in Software Engineering Courses, *Journal of Communications Software and Systems*, 17(4), pp.342–349. doi: 10.24138/jcomss-2021-0046.

Shynkarenko, V., Zhevaho, O. (2024). A Video-Based Approach to Learning Debugging Techniques. *14th International Scientific and Practical Programming Conference, UkrPROG 2024. CEUR Workshop Proceedings*, 3806, pp. 462–473. Available at https://ceur-ws.org/Vol-3806/S_18_Shynkarenko_Zhevaho.pdf.

Skalozub, V., Ilman, V., Shynkarenko, V. (2017). Development of ontological support of constructive-synthesizing modeling of information systems, *Eastern-European Journal of Enterprise Technologies*, 6(4 (90)), pp.58–69. doi: 10.15587/1729-4061.2017.119497.

Wachter, H., Michaeli, T. (2024). Analyzing Teachers' Diagnostic and Intervention Processes in Debugging Using Video Vignettes, *17th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, 15228, pp.167–179. doi: 10.1007/978-3-031-73474-8_13.

Whalley, J., Settle, A., Luxton-Reilly, A. (2021). Analysis of a Process for Introductory Debugging, *23rd Australasian Computing Education Conference*, pp.11–20. doi: 10.1145/3441636.3442300.

Wilkin, G. A. (2025). "Debugging: From Art to Science" A Case Study on a Debugging Course and Its Impact on Student Performance and Confidence, *56th Annual SIGCSE Technical Symposium on Computer Science Education*, pp.1225–1231. doi: 10.1145/3641554.3701893.

Yamamoto, R., Noguchi, Y., Kogure S., Yamashita, K., Konishi, T., Itoh, Y. (2016). Design of a learning support system and lecture to teach systematic debugging to novice programmers, *24th International Conference on Computers in Education: Think Global Act Local*. doi: 10.58459/icce.2016.1178.

Yang, S., Baird. M., O'Rourke, E., Brennan, K., Schneider, B. (2024). Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions, *ACM Transactions on Computing Education*, 24(4), pp.1–44. doi: 10.1145/3690652.

Zhang, Y., Paquette, L., Pinto, J. D., Liu, Q., Fan, A. X. (2022). Combining latent profile analysis and programming traces to understand novices' differences in debugging, *Education and Information Technologies*, 28(4), pp.4673–4701. doi: 10.1007/s10639-022-11343-7.

Received April 27, 2025, revised July 1, 2025, accepted July 19, 2025