Slicer4D: A Slicing-based Debugger for Java

Sahar Badihi
Univ. of British Columbia, Canada
shrbadihi@ece.ubc.ca

Sami Nourji
Univ. of British Columbia, Canada
saminourji23@gmail.com

Julia Rubin
Univ. of British Columbia, Canada
mjulia@ece.ubc.ca

Abstract

Debugging software failures often demands significant time and effort. Program slicing is a technique that can help developers fast track the debugging process by allowing them to focus only on the code relevant to the failure. However, despite the effectiveness of slicing, these techniques are not integrated into modern IDEs. Instead, most, if not all, current slicing tools are launched from the command line and produce log files as output. Developers thus have to switch between the IDE and command line tools, manually correlating the log file results with their source code, which hinders the adoption of the slicing-based debugging approaches in practice.

To address this challenge, we developed a plugin extending the debugger of IntelliJ IDEA – one of the most popular IDEs – with slicing capabilities. We named our slicing-based debugger extension Slicer4D. Slicer4D offers a user-friendly interface for developers to perform dynamic slicing and further enhances the debugging experience by focusing the developers' attention only on the parts of the code relevant to the failure. Additionally, Slicer4D is designed in an extensible way, to support integration of a variety of slicing techniques. We hope our tool will pave the way to enhancing developer productivity by seamlessly incorporating dynamic slicing into a familiar development environment.

Tool implementation and evaluation package is online [16].

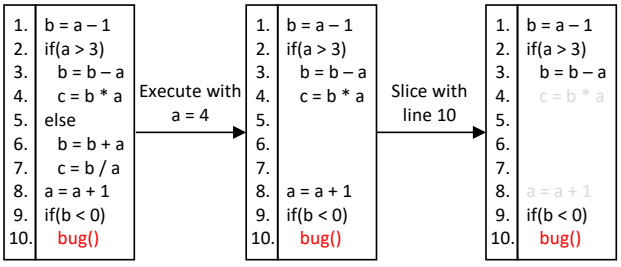
Keywords

Program analysis, dynamic slicing, Java, debugging, IntelliJ plugin

ACM Reference Format:
Sahar Badihi, Sami Nourji, and Julia Rubin. 2024. Slicer4D: A Slicing-based Debugger for Java. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE'24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3691620.3695363

1 Introduction

In modern software development, debugging is crucial in identifying and rectifying failures. However, debugging complex programs can be a challenging task, often requiring developers to navigate through a vast code base. To recover reasons for a failure, it is estimated that developers spend up to 50% of their debugging time building mental models of control and data flow dependencies of the code leading to the failure [15]. This process is often time-consuming and error-prone.



(a) Source code. **(b) Executed code.** **(c) Sliced code.**

Figure 1: Dynamic slicing example.

Dynamic program slicing [11] is a technique positioned to make debugging tasks more efficient. Through automatically extracting control and data flow dependencies of a particular variable or statement of interest, it allows developers to focus only on the executed statements relevant to the failures, minimizing the amount of code developers need to inspect [2].

For example, consider the code snippet in Figure 1a, which has a bug in line 10. This code gets the variable "a" as input in line 1 and then performs a set of calculations based on this variable. Figure 1b shows the subset of statements executed when a = 4, omitting the non-executed statements in lines 5-7. Figure 1c further grays out executed statements that are not in the slice and thus are not relevant to the bug (lines 4 and 8). In this case, only the calculations related to the variable "b" affect the "if" statement in line 9, which, in turn, determines whether the buggy line is executed or not. See Section 2.2 for a more formal definition of dynamic slicing.

A number of existing implementations of the classic dynamic slicing algorithm optimize for different trade-offs between accuracy and performance. JavaSlicer [6] and Slicer4J [3] are examples of publicly available implementations for Java. In addition to the classic slicing implementations, a number of slicing variants, such as chopping [9], dicing [18], barrier slicings [13], and thin slicing [17], aim to minimize the size of the slice while ensuring it still contains the information relevant to the failure. Some of these techniques require extra input(s) from developers. For example, *barrier slicing* asks for a set of statements, called *barriers*, which are assumed to be bug-free. When computing the slice, this technique excludes from the slice the barriers and all statements affecting the barriers to make the slice more focused.

While these techniques are effective in selecting trace statements related to the fault, most, if not all of them, provide only a command-line interface and log-file outputs. This hinders the adoption of slicing in practice, as developers rather need the slice-based support to be integrated into their current workflows, to debug their code "natively" and consistently with their working practices.

ASE'24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695363

To bridge the gap between slicing research and practical usage, we contribute a plugin named Slicer4D. The plugin extends one of the most popular contemporary IDEs, IntelliJ IDEA [7]. Slicer4D allows IDE users to perform slicing using the GUI and further integrates the slice output into the development workflow. Specifically, it grays-out those code sections in the IDE that are not included in the slice, augments the debugger to only focus on statements in the slice when stepping, and shows only variables relevant to the slice in the variable inspection window. To support a variety of different slicing techniques, Slicer4D is designed in a modular way; it provides an extendability interface which can be leveraged by third-party slicing techniques in order to integrate their slicing output into the IDE. Its out-of-the-box version also includes an integration with Slicer4J [3]. The implementation and evaluation package of Slicer4D are available online [16].

2 Background

We now provide an overview of debugging capabilities implemented in typical IDEs and formally define dynamic slicing.

2.1 IDE Support for Debugging

Most IDEs, such as IntelliJ, Visual Studio Code, and Eclipse, have integrated debugging support, where a developer can (1) specify points in the program where the execution should be suspended, called *breakpoints*, (2) step through the code and method invocations, and (3) inspect variables and expressions. This exploration process helps developers gain knowledge about the causes of failures, facilitating their fixes.

2.2 Dynamic Slicing

When a program runs, each statement can be triggered multiple times during the execution, e.g., in multiple iterations of a loop or in different instances of a thread. We refer to each individual execution of a statement as a *statement instance* and denote the k^{th} execution of a statement s_i as s_i^k. We refer to the full sequence of statement instances from a particular app run as an *execution trace*. For the example in Figure 1, we refer to a statement in line i as s_i, e.g., the `if` statement in line 2 is denoted by s_2. The execution trace for this example thus contains statement instances s_1^1, s_2^1, s_3^1, s_4^1, s_8^1, s_9^1, and s_{10}^1 (Figure 1b).

A statement instance s_j^m is *control-dependent* on s_i^k if s_i^k can alter the program's control and determines whether s_j^m executes [5]. In Figure 1, s_3^1 is control-dependent on s_2^1, as the outcome of the `if` determines whether the control reaches s_3^1 or not. A statement instance s_j^m is *data-flow-dependent* on s_i^k w.r.t. the variable v used in s_j^m if and only if s_i^k defines v and no other statement redefines v between s_i^k and s_j^m in the trace [1]. In Figure 1, s_3^1 is data-flow-dependent on s_1^1 w.r.t. `b`, as the statement in line 1 defines the variable `b` used in line 3.

A *slicing criterion* for an execution trace is a statement instance and all variables of interest used in this statement instance [11]. A *backward dynamic slice* [11] is a set of statement instances on which the slicing criterion is control- or data-flow-dependent, either directly or transitively. For the example in Figure 1, the backward slice from s_{10}^1 consist of statement instances s_9^1, s_3^1, s_2^1, and s_1^1.

3 Slicer4D Design

Figure 2 shows an overview of Slicer4D. The plugin is designed to be modular and extensible: it decouples the slicing logic, the UI, and the debugging modifications so that it can handle implementations of multiple different slicers. At a high level, it includes our *IDE* extensions to support the integration of new slicers, their setup, and the debugging functionality itself. The *API Layer* specifies a set of interfaces that *Slicer Providers* should realize. Integration of slicers is supported via a wrapper that serves as a liaison between the slicing tool and our IDE extensions. We now describe these parts in more detail.

IDE extensions serve as the primary interface for both slicer providers and users. They have three key parts, each tailored to a specific task:

- **Extendability** allows a slicer provider to integrate a new slicing technique, enabling the addition of custom slicers to the IDE.
- **Setup** allows a slicer user to select a slicing technique from available slicers that are already registered with the plugin. The slicer user can also select a slicing criterion and provide additional parameters required by custom slicers.
- **Debugging** allows a slicer user to utilize the enhanced debugging functionalities using the information generated by a slicing technique. After getting the slice, the IDE code window highlights the slice statements and grays out the statements that are not in the slice. The IDE also enables breakpoints only for statements included in the slice and modifies the stepping commands to traverse only statements in the slice. For example, the *step over* command jumps to the next statement in the slice rather than the next statement in the execution, as the original *step over* command would do. Similarly, the modified variable window displays only the variables included in the slice and removes other variables shown in the original variable window.

API Layer provides a standardized communication interface between the IDE and slicer wrappers through the APIs below:

- *List<ParameterSpec> getConfiguration()*: gets a list of parameter specifications required by a slicer. The specification includes a name, extension point (code editor, debugger editor, etc.), and parameter type (statement or variable). This interface is used when a slicer provider wants to integrate their new slicer into the IDE.
- *boolean setSlicingCriterion(Statement, List<Variable>)*: sets slicing criteria, which is a statement and a subset of its variables, as specified by the user. It returns *true* if the slicer succeeds in setting the criterion and *false* otherwise.
- *boolean setParameters(Map<ParameterSpec, List<Value>>)*: sets the values (statements or variables) specified by the slicer user for each parameter.
- *boolean isInSlice(Statement)*: checks if a given statement is in the slice. This API is used to highlight the statements in the code view and also enable the breakpoint toggling for slice statements.
- *Statement nextInSlice(Statement)*: given an in-slice statement as an input, it retrieves the next statement in the slice. This API is used to manage the modified stepping commands to skip the next statement if it is not in the slice.
- *Statement prevInSlice(Statement)*: given an in-slice statement as an input, it retrieves the previous statement in the slice.

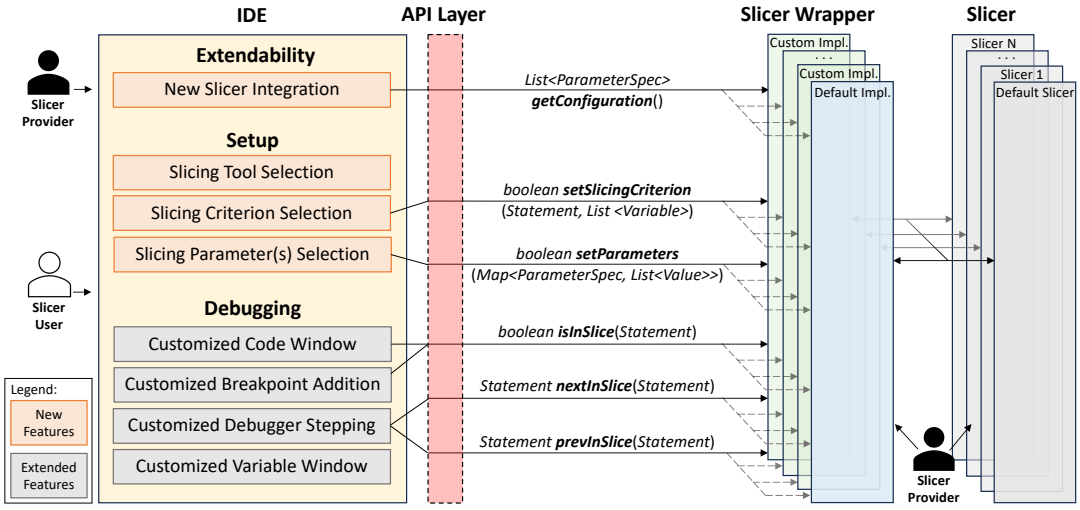


Figure 2: Slicer4D overview.

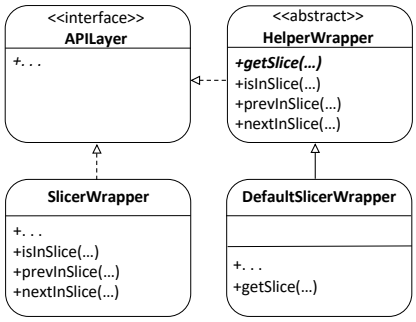


Figure 3: Slicer wrapper class diagram.

Slicer Wrapper supports integration of custom slicers. It serves as a liaison between the slicing tool and our IDE. Figure 3 provides more details on the implementation of wrappers. A *SlicerWrapper* has to implement all *APILayer* interfaces. This can be done either directly or by extending the *HelperWrapper* abstract class we provide. The *HelperWrapper* class reads and translates a simple slice log to the format required by the IDE. It relies on and uses the *getSlice()* method, which obtains the slice log from the slice provider, to realize the remaining methods, such as *isInSlice()* and *nextInSlice()*. Thus, wrappers extending *HelperWrapper* only have to implement *getSlice()* instead of three slicer navigation methods. This provides more flexibility in integrating third-party slicers not originally designed to work with Slicer4D. As one example, we introduce a default wrapper for the Slicer4J [3] integration.

4 Preliminary Evaluation

We investigated the usefulness of Slicer4D in a preliminary study.

Methodology. We conducted a user study involving 10 upper-level (fourth- and fifth-year) undergraduate students in our university. All participants had prior experience with IntelliJ IDEA and debugging Java programs using this IDE: three were proficient, six intermediate, and one identified themselves as a beginner. Each participant had completed at least one co-op term of four months working as a software developer in a company. The age of the participants ranges between 21 and 26 years, with the majority being 23 years old. As the study subjects, we selected three Java programs (30 LoC, on average) and injected a bug into them. We provided each participant with a randomly selected subject and asked them to use Slicer4D to debug the program. Participants were then asked to answer the following questions:

- How useful do you find Slicer4D (10-point scale, 1=not useful at all, 10=very useful)?
- Which of Slicer4D's features did you like the most?
- Which of Slicer4D's features did you dislike the most?
- Any particular features that you want to see added?
- How easy is it to learn how to use Slicer4D (10-point scale, 1=not easy at all, 10=very easy)?

To avoid biasing the participants, we recruited two students not associated with our research group to conduct and moderate the study. The moderators used their computers to allow the participants to access a running version of Slicer4D. Participants were informed of the study's purpose and were briefed on what was required of them during the study, including debugging a program and completing a retrospective survey.

Results. Overall, the participants expressed a high level of satisfaction with Slicer4D (average usefulness of 8.6/10 and ease to learn of 8/10). Qualitative feedback was also positive, with participants pointing to Slicer4D's intuitive interface and the clarity of the slicing visualizations. In particular, they were most satisfied with the *line graying* functionality, followed by the modified *stepping* commands. Some participants mentioned that the plugin helped them with "decreasing debugger complexity" as it "shows a smaller program to step over".

As suggestions for improvements, participants pointed out the lack of interactive visualizations (e.g., graphs), which made it difficult for them to understand why certain statements were included in the slice and what control and data flow dependencies existed between the slice statements. One participant suggested highlighting lines of code with different colors based on their dependency type, i.e., control and data. Participants also noticed delays in starting

the debugging session due to the time needed for the underlying slicing technique to finish execution. A visual progress bar could help improve the user experience related to this issue. Finally, participants suggested adding visuals to indicate which statements were selected as slicing criteria/parameters.

Overall, despite these suggestions for improvements, the study suggests that participants had an overall positive experience using Slicer4D in the debugging process.

5 Limitations, Discussion, and Future Work

In its current version, Slicer4D only supports two types of additional parameters provided to custom slicers: *statements* and *variables*. While these are the parameters required by the contemporary popular slicing techniques, Slicer4D can be further extended to support other types of parameters that might become necessary in the future. In addition, while the Slicer4D's architecture is not tied specifically to IntelliJ, its current implementation extends functionality within this IDE. The usability of the Slicer4D can further be improved based on the feedback obtained in the user study. Finally, our user study involved a relatively small sample of student participants from a single university, which may limit its generalizability to the broader developer community.

Despite the limitations, we believe Slicer4D presents promising opportunities for future development and community collaboration. Our implementation is openly available and we encourage the community to join our effort and further improve the tool. We also encourage slicer providers to integrate their slicing tools into Slicer4D, helping to broaden its applicability and usefulness. The feedback from a more diverse set of users and the slicer provider community, which we hope to achieve through this tool publication and demonstration, will be invaluable in refining and further enhancing the tool.

As an additional way forward, we are also exploring the integration of Slicer4D into continuous integration (CI) pipelines, to use it with slicing techniques for regression analysis. This would enable more automated and efficient debugging workflows for evolving large-scale software projects.

6 Related Work

Several prior works have focused on enhancing program debugging by integrating various analyses into IDEs. MagpieBridge [14] is a general framework for integrating static analyses into IDEs, providing developers with immediate feedback on potential issues in their code. IntelliJ IDEA also includes a built-in static analysis for tracing data transformations [8]. However, static analysis techniques are less accurate for debugging failures due to over-approximation static analysis makes.

Some approaches aim to provide extra information about the current program execution, e.g., by displaying the call stack as a UML sequence diagram [4] or displaying variables as a UML object diagram [12]. While these approaches are sometimes referred to as "visual debugger helpers", our work is orthogonal and complementary: we focus on integrating slicing into IDE by identifying statements relevant to the bug and modifying the code inspection and debugging processes to consider only these statements. Nevertheless, Slicer4D can be further augmented with these visualization ideas, to display information generated by slicing.

Several standalone debugging aids, e.g., Whyline [10], aim at helping developers understand the software behavior by allowing the developers to ask "why" and "why not" questions about program behaviors. These tools are not integrated into the IDEs, whereas Slicer4D focuses on IDE integration.

7 Conclusions

We introduced Slicer4D, an IntelliJ plugin that integrates slicing techniques into the debugging workflow, i.e., code inspection, breakpoints, and stepping. Slicer4D is modular and extensible, separating the slicing logic from the user interface and debugging modifications. This design makes it possible for the IDE to interact with multiple slicers through a well-defined API layer, ensuring flexibility and scalability. Our preliminary evaluation showed high satisfaction with Slicer4D's interface, despite areas for improvement. We hope that our work will provide a platform for integrating slicing techniques into standard workflows used by developers.

Acknowledgments. We would like to thank UBC capstone teams for their contributions to the early versions of this plugin.

References

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. 1991. Dynamic Slicing in the Presence of Unconstrained Pointers. In *Proc. of the Symposium on Testing, Analysis, and Verification (TAV)*. 60–73. https://doi.org/10.1145/120807.120813

[2] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. *ACM SIGPLAN Notices* 25, 6 (1990), 246–256. https://doi.org/10.1145/93542.93576

[3] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Slicer4J: A Dynamic Slicer for Java. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[4] Jeffrey K Czyz and Bharat Jayaraman. 2007. Declarative and Visual Debugging in Eclipse. In *Proc. of the International Conference on Object-oriented Programming, Systems, Languages, and Applications Workshop on Eclipse Technology eXchange (OOPSLA)*. 31–35.

[5] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[6] Clemens Hammacher. 2008. Design and Implementation of an Efficient Dynamic Slicer for Java. Bachelor's Thesis.

[7] IntelliJ IDEA. 2021. https://www.jetbrains.com/idea/

[8] IntelliJ. 2024. Data Flow Analysis. https://www.jetbrains.com/help/idea/analyzing-data-flow.html

[9] Daniel Jackson and Eugene Joseph Rollins. 1994. *Chopping: A Generalization of Slicing*. Carnegie-Mellon University. Department of Computer Science.

[10] Amy J Ko and Brad A Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proc. of the International Conference on Software Engineering (ICSE)*. 301–310.

[11] Bogdan Korel and Janusz Laski. 1988. Dynamic Program Slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–163. https://doi.org/10.1016/0020-0190(88)90054-3

[12] Tim Kräuter, Harald König, Adrian Rutle, and Yngve Lamo. 2022. The Visual Debugger Tool. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 494–498.

[13] Jens Krinke. 2003. Barrier Slicing and Chopping. In *Proc. of IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*. 81–87.

[14] Linghui Luo, Julian Dolby, and Eric Bodden. 2019. Magpiebridge: A General Approach to Integrating Static Analyses into IDEs and Editors. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*.

[15] Anneliese Mayrhauser and A. Marie Vans. 1997. Program Understanding Behavior During Debugging of Large Scale Software. *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers* (10 1997), 157–179. https://doi.org/10.1145/266399.266414

[16] ReSeSS. 2024. Slicer4D. https://github.com/resess/Slicer4D

[17] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. 112–122.

[18] Mark Weiser and Jim Lyle. 1986. Experiments on Slicing-based Debugging Aids. In *Proc. of the Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. 187–197.