

The Slice Navigator: Focused Debugging with Interactive Dynamic Slicing

Arian Treffer
Hasso-Plattner-Institut
Potsdam, Germany
Email: arian.treffer@hpi.de

Matthias Uflacker
Hasso-Plattner-Institut
Potsdam, Germany
Email: matthias.uflacker@hpi.de

Abstract—Slicing is a technique to reduce the amount of code that needs to be analyzed for a given problem.

In this paper, we present the Slice Navigator, a debugging tool for Java programs that combines dynamic slicing with back-in-time debugging data to support the debugging process in multiple ways. Firstly, it supports the developer’s short term memory by providing a summary of relevant program state and context for the current instruction. Secondly, it provides an alternative to breakpoints as it can be used to control the debugger to jump to related instructions, such as the last change of a variable. Thirdly, it allows to directly reconfigure the slicing criteria, enabling the developer to minimize the search space of active code without interrupting the debugging workflow.

The paper focuses on the UI and implementation of the Slice Navigator view and how it changes the debugging workflow. A performance evaluation shows the feasibility of our approach for larger programs.

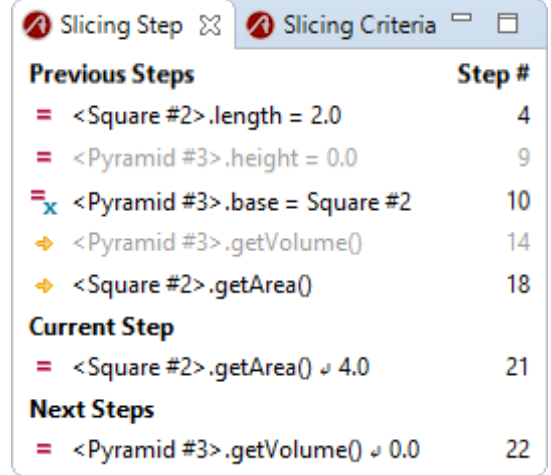
I. INTRODUCTION

In many cases, software bugs don’t cause the software to fail, i.e., to deviate from expected behavior, immediately. To actually find the bug, developers have to follow the chain of erroneous state from the observed failure backwards to the bug. Many approaches exist to support this process.

Debuggers allow to inspect the state of a running program and to understand its impact on the program’s behavior. Back-in-time, or “omniscient” debuggers (ODBs) even make it possible to follow the infection chain backwards through time, removing the overhead of frequently restarting the debug session [1]. However, developers still need to manually identify the relation between states without spending too much time in irrelevant parts of code. This often requires a high familiarity with the code, which is not always given.

Weiser has shown that programmers think not only in modules, but in related statements [2]. Slicing is a technique to produce subsets of a program relevant to a given criterion. Dynamic slicing also considers the program input, which allows to remove even more irrelevant instructions [3].

Slicing suffers from a similar problem as debugging: every time the developer’s question changes the slice has to be recomputed, which can interrupt the developer’s flow



Slicing Step		Slicing Criteria	
Previous Steps			Step #
=	<Square #2>.length = 2.0		4
=	<Pyramid #3>.height = 0.0		9
=x	<Pyramid #3>.base = Square #2		10
◆	<Pyramid #3>.getVolume()		14
◆	<Square #2>.getArea()		18
Current Step			
=	<Square #2>.getArea() ↗ 4.0		21
Next Steps			
=	<Pyramid #3>.getVolume() ↗ 0.0		22

Figure 1: The Slice Navigator shows context for the current debug step. Steps directly related to the current step are listed in black, steps with overarching dependencies in gray. Small icons indicate how the steps are related.

even if it only takes a few seconds. Furthermore, every time developers need to switch between slicer and debugger, another interruption occurs. This tool barrier discourages many developers from using slicing at all.

It has been shown that slicing can be useful to improve developer productivity [2], [3], especially for developers dealing with very complex or unfamiliar code.

We present a new approach that combines omniscient debugging and dynamic slicing. While developers omnisciently debug a dynamic slice, at any point they can add or adjust the slicing criteria and changes are applied instantly, without interrupting the debug session. A new UI component, the Slice Navigator, provides a unique view on the execution by combining relevant information from both the ODB and the slicing subsystem.

The contributions of this paper are threefold:

- A new dynamic slicing algorithm allows quick and iterative refinement of the slicing criteria to adapt the slice to changing developer questions. Based on previous work, developers can formulate their question by choosing from different dependency types that will

change the outcome of the slice [4].

- The *Slice Navigator* is a UI component that bundles access to the debugger and the slicer. It provides context for the current instruction by showing relevant parts of the slice, allows developers to iteratively refine the slicing criteria, and serves as an alternative to breakpoints and stepping.
- The integration of dynamic analyses directly in the debugger not only reduces interruptions in developer flow by minimizing context switches between tools and shortens waiting time as recorded run-time data can be re-used; it also allows for a new debugging workflow where developers can isolate a bug by iteratively slicing away correct code.

The remainder of this paper is structured as follows: The next section summarizes related work. Section III explains the Slice Navigator user interface and how it allows for an improved debugging workflow. Section IV discusses relevant details of the implementation and the slicing algorithm. Section V evaluates the feasibility of our approach. Section VI concludes.

II. RELATED WORK

The concept of slicing was introduced by Weiser and initially only built on static analysis [2], [5]. Later, Korel and Laski [6] and Agrawal and Horgan [3] introduced dynamic slicing by proposing different ways of extending the concept to the run-time.

The idea of being able to debug backwards in time dates back to the 1960s with EXDAMS, a debugger for FORTRAN [7]. Since then, debuggers with back-in-time capabilities have been implemented for many programming languages [8], [9], [10].

In 2003, Lewis introduced the concept of “omniscient debugging”, a debugger that can not only rewind time, but has instant access to every point in past and future [1]. Later work on omniscient debugging focused mostly on handling the large amounts of data such a debugger creates [11], [12].

Perscheid et al. combined omniscient debugging and dynamic code analysis similar to slicing to automatically identify likely locations for code defects by analyzing a suite of unit tests with passing and failing test cases [13].

SPYDER is a debugger with slicing and back-and-time capabilities [8]. With SPYDER, Agrawal et al. also proposed a new debugging workflow that is similar to our approach. However, while Agrawal et al. proposed to create a new slice every time the programmers question changed, our approach focuses on iteratively modifying a single slice. Furthermore, we present a new way of presenting aspects of the slice to developers.

III. A NEW DEBUGGING WORKFLOW

Both omniscient debugging and (dynamic) slicing changed the way how developers approach fault localization. In this section, we use a simple example to

demonstrate how we integrated existing and new tools to an improved debugging workflow. We also describe the user interface of the Slice Navigator, how it presents information about the slice, and how it can be used to control the debugger.

A. Getting Started

Very often, the starting point for a debug session is a reproducible observable program failure, preferably in the form of a failing unit test. Using an omniscient debugger, developers halt the execution at the failing line of code to observe the program state. From here, they want to back-track the erroneous state. However, they quickly realize that the code contains many other side effects making it hard to follow the state of interest.

Using a pure omniscient debugger, developers would now have to track the relation of states to identify the infection chain. In other words, they have to manually create a dynamic dependency graph using only their short-term memory.

Using our approach, the developer right-clicks the erroneous state in the debugger’s variables view and chooses slicing from the context menu. This will set the variable or field as a slicing criterion and start the slice computation.

Once the slice is computed, all debugging views (e.g., the trace and the variable view) will show instructions or variables not belonging to the slice only in gray. Stepping through the execution will skip instructions not belonging to the slice.

We will use a small code example to explain the user interface and internals of the slice navigator. Listing 1 shows two Java classes and a failing JUnit test case. In our scenario, after noticing a failed test case, the developer chooses to slice on the arguments of the `assertEquals` invocation in line 29. Because we are using only a minimal example, the resulting slice contains almost the entire program. When we tested the Slice Navigator on real open source programs, this step often already removed a lot of code.

Nevertheless, for a complex program the initial slice can still be too large to allow an efficient search for the problem. In this case, the developer can now use the slice navigator to get an overview of the execution and to iteratively adjust the slicing criteria.

B. The Slice Navigator

The first purpose of the slice navigator is to aid the developer’s short-term memory. It provides a quick overview over previous and upcoming events, and how they relate to the current instruction. Figure 1 shows a screenshot of the slice navigator with the execution of the example test-case halted on the return instruction of `getArea()` in line 7.

A step, or event, is any instruction that has a side effect on the program state. “Previous Steps” lists all past events that the current or future events depend on.

```

1 class Square implements Shape2D {
    private double length;
    public Square(double length) { this.length = length; }

5    @Override
    public double getArea() {
        return length * length;
    }
}

10 class Pyramid implements Shape3D {
    private Shape2D base;
    private double height;
    public Pyramid(Shape2D baseShape, double height) {
15        base = baseShape;
        height = height;
    }

    @Override
    public double getVolume() {
20        return base.getArea() * height / 3;
    }
}

25 class PyramidTest {
    @Test
    public void test_getVolume() {
        Shape3D shape = new Pyramid(new Square(2), 6);
        assertEquals(8, shape.getVolume());
30    }
}

```

Listing 1: Example program with a failing test case

Likewise, “Next Steps” shows all events that depend on the current or previous events.

If a step is shown in black, it has a direct dependency link to the current step. Steps shown in gray have dependencies that go beyond the current step. I.e., gray “previous steps” have dependency links to “next steps”, and vice versa.

Simply by looking at the previous and next steps, the developer can understand at a glance how the current instruction fits into the greater scheme. This is particularly useful if the current instruction was reached via a breakpoint, in which case it is not always obvious at which point in time it was hit.

To obtain this kind of information with a regular debugger, developers need to analyze the execution stack and maybe even inspect lower stack frames. But even then it is not always obvious which part of the program state that is still reachable is actually used again. Unlike a typical debugger’s variable view, the Slice Navigator only shows relevant variables, and also shows relevant object fields on the first level.

Further, the slice navigator shows details about the dependency graph that was used to compute the current slice. Small icons indicate how the events of the slice are related. The meaning of these symbols will be explained in the next subsection.

Debugging with the Slice Navigator is simple: To investigate the origin of a value, developers can simply click

Slicing Step		Slicing Criteria	
Previous Steps		Step #	
=	<Pyramid #3>.height = 0.0		9
↗	<Pyramid #3>.getVolume()		14
Current Step			
=	<Pyramid #3>.getVolume() ↘ 0.0		22
Next Steps			
✂	actual = 0.0		23

Figure 2: The program halted at line 21, after `getArea()` was removed from the slice.

on it to move the execution to that point in time. This way, the slice navigator allows to efficiently follow infection chains of erroneous state. Likewise, it is easy to follow up on the impact of an instruction by navigating to its future dependencies.

However, using the Slice Navigator developers can not only control the debugger, they can also adjust the slicing criteria.

C. Interactive Slice Configuration

The slicing component is based on previous work [4]. When building the dependency graph between instructions, the algorithm distinguishes between three types of dependencies.

Value dependencies occur when the the value of an instruction is derived from another instruction’s value. In the slice navigator, they are represented with a red equality sign.

Instructions that determine if another instruction can be reached are *reachability dependencies*, indicated by a yellow arrow. Typically, these are method invocations and instruction in conditional statements.

Sometimes, a value depends on only one of multiple candidate values. A *control dependency* determines which of these candidates is used. More formally, control dependencies are reachability dependencies of value candidates that are not also reachability dependencies of all other candidates. In the navigator, they are indicated by a blue “X”.

Developers can now combine these different dependency types to adjust the slice for specific purposes. Clicking on an event’s dependency symbol brings up a dialog that allows to choose which dependencies of that event to include. This way developers can, for instance, put a focus on how a value was computed or how an instruction was reached. It is also possible to remove all dependencies of an event, for instance if it is known to be correct and its history is not of interest, thereby moving the focus of the slice to less well-understood parts of the program.

Whenever a slicing criterion is modified, the slice is updated instantly, without locking the user interface or

resetting the current debug session. In our examples, developers might choose to exclude the result of `getArea()` from the slice as it is correct. As shown in figure 2, with the computation of the area removed from the slice it is now much easier to see that the wrong result of `getVolume()` was caused by a wrong value in height.

As mentioned before, instructions and states not belonging to the slice are still shown in the IDE, mostly to serve as an orientation help, to provide context to the current operations. However, it might also happen that a value or instruction outside of the slice catches a developers attention. In this case, they can choose to add it as another slicing criterion and the slice is immediately expanded. Again, this happens without interrupting the developer's work.

IV. IMPLEMENTATION

The Slice Navigator is part of a set of debugging tools that we implemented as a plug-in for the Eclipse IDE.

A. Framework

The high-level architecture of our prototype consists of three components: the tracer, the event database, and the omniscient debugger including the slicing module.

The tracer is implemented as a Java agent that modifies the bytecode of a program to insert tracing instructions that log all events into csv files. With our Eclipse plug-in, the developer can initiate an omniscient debug session by selecting a customized launcher in the run configuration. The launcher will add VM arguments to the execution to configure the tracer. Once the execution completes, the launcher will automatically import the trace data into a previously configured database. We provide launchers for basic Java applications and for JUnit test suites. For JUnit launches, the tracer will treat each test case as an independent execution and ignore code of the testing framework.

The database stores all events of a set of executions. It is possible to set up one database per project or one for the entire workspace. We currently support HSQLDB, MySQL, and SAP Hana, but in principle any relational database can be used.

The omniscient debugger consists of a set of views that allow to debug a Java application based on the data from the database. It is a post-mortem debugger, i.e., it simulates a debug session while the actual program has already terminated.

B. Initial slice computation

The basic structure is of the algorithm for building the initial slice is fairly simple and shown in listing 2.

The slicing criterion is a set of events, the slice is initially an empty set. For each event that is added to the slice, the static dependency graph of the event's method is obtained.

If the graph is not initialized yet, it will be created by a code analysis using the Soot framework¹. The dependency

```

1 function build_slice(criteria)
  slice ← ∅
  for each event ∈ criteria do in parallel
    add_to_slice(slice, event)
5 return slice

function add_to_slice(slice, event)
  if event ∈ slice ∨ event.is_negative_criterion then return
  slice += {event}
10 static_dependency_graph ← event.method.dependency_graph
  static_dependencies ← static_dependency_graph.get(
    ↪ event.instruction, event.dependency_flags)
  for each instruction ∈ static_dependencies do in parallel
    prev_event ← last_previous_event(event, instruction)
    if prev_event exists then
15 prev_event.inherit_dependency_flags(event)
    add_to_slice(slice, prev_event)

```

Listing 2: Simplified algorithm for building the slice

graph contains every instruction's static dependencies and distinguishes between three dependency types, as described in the previous section. Dependency instruction candidates are looked up in the graph by the event's instruction.

For each candidate instruction, the latest occurrence is looked up in the event database. For variable assignments the look up is limited to the current method invocation, for events like field assignments the scope is not limited. If an event was found, it will be added to the slice next.

Every event carries flags that indicate which types of its dependencies the slice should include. If the dependency flags are empty, the event is considered a negative criterion and is removed entirely from the slice (cf. line 8). If the dependency flags of an event were not configured directly by the user, they are inherited from the depending events (cf. line 15).

As can be seen, the algorithm is designed to follow all dependency links concurrently. This allows to use parallelization to improve the performance of the slicer. The amount of concurrency is only limited by the theoretical maximum level of concurrency in the original program. This is to be expected as the slicer is, in some sense, just evaluating the program backwards.

However, we also let the slicing run concurrently to the user interface. Twice per second, we update the different debugging views, such as variable and trace explorer, to use the latest intermediate result. This way, we not only don't have to freeze the user interface, but we can allow developers to continue working even if computing the whole slice takes several seconds.

C. Incremental slicing

Adding another event to the slicing criteria works similarly to building a new slice, the major difference being that the slice is not initially empty.

When developers want to remove an event from the slice, we need to know when to cascade the removal to its dependencies. As the dependency graph is acyclic, we can

¹<https://sable.github.io/soot/>

use reference counting for this. However, we need to independently count the references for each dependency type. When the counter for one of the dependency types reaches zero, dependencies of that type are unlinked respectively.

Changing the dependency type of an event, e.g., from value to reachability, is a combination of removing and adding an event.

D. Filling the Slice Navigator

To fill the Slice Navigator, we iterate over the events in the slice, ordered by step number, beginning at the current step, and collect all events that will be shown in the view. Events directly related to the current step will be flagged as such, so that they can be highlighted.

Events at the current step and their dependencies are always collected. For events in the future, it is first determined if they have direct dependencies that are in the past. If such dependencies exist, they are collected together with their respective future event.

However, our collection algorithm considers two special cases:

First, reachability dependencies are not collected except for the current step. This measure is taken to prevent the entire method stack from appearing in “previous steps” and all future variable changes in “next steps”.

The second special case are transparent classes. Classes of the JDK are flagged as transparent, and all their internal events, i.e., all events except direct accesses from non-transparent classes, are resolved immediately and never appear in the navigator.

V. EVALUATION

One of the main advantages of the Slice Navigator is that it integrates into the debugging workflow. As such, it is crucial that results are produced in a timely manner, as a waiting time of even a few seconds may interrupt developers in their flow.

To evaluate the performance of our approach on real-world code, we measured the computation of several slices on JUnit tests of an open-source business-process engine². All tests were run on a 2.0 GHz Intel i7 Processor with 4 cores and 8 GB RAM, running Windows 8.1. A MySQL database was used to store the trace data. We repeated every measurement 10 times, all charts show the average values.

We previously observed that our approach differs from other slicing implementations insofar as that the run-time of the algorithm does not depend on the total length or run-time of the sliced program, but only on the size of the resulting slice [4]. Our current measurements confirm that slicing time is linear to the result size.

Figure 3 shows the time for computing the initial slice in seconds, depending on size of the resulting slice. Times are given in total, and divided into static code analysis and the dynamic analysis of the event data.

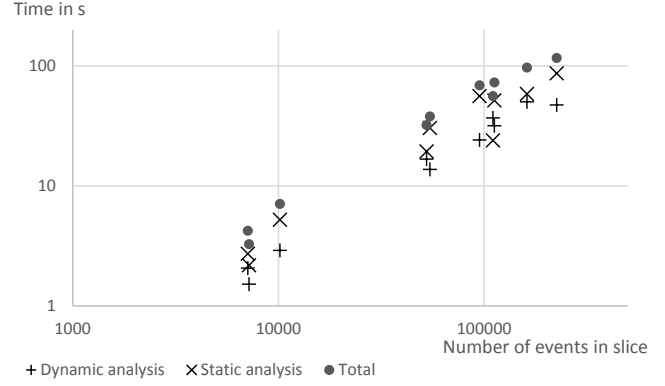


Figure 3: Time for computing the initial slice

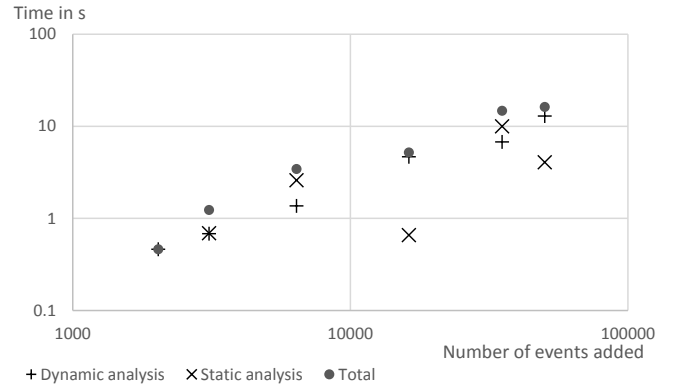


Figure 4: Time for adding a slicing criterion

As can be seen, the static analysis takes slightly more time on average. It should be noted that the total time is less than the sum of the static and dynamic analysis times, as both can, to some extent, run in parallel. The chart shows that in our setup the algorithm was able to process approximately 1000 events per second.

When adding additional elements to the slice, previously computed dependency graphs can be reused. As figure 4 shows, the time for dynamic analysis remains constant per event. The time for static analysis, on the other hand, shows great variation and depends on how much new code was included by the broadened slicing criteria. In the worst case, expanding the slice takes as long as creating a new slice for only that event.

Figure 5 shows that removing events from the slice by narrowing the slicing criteria is significantly faster, as no actual analysis has to be performed. Likewise, adding those events again by reverting the slicing criteria change is fast, as previously computed dependency graphs can be reused.

From the results in figure 3, it seems as if the slicing algorithm is too slow to be of practical use to a developer. A single second of execution can produce several hundreds

²<https://github.com/camunda/camunda-bpm-platform>

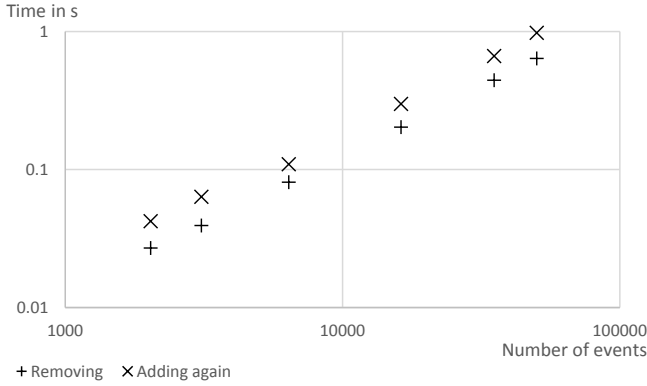


Figure 5: Time for removing events from the slice

of thousands of events and it is generally not feasible to wait multiple minutes for the slicing to complete. However, due to parallelization and the way the algorithm works, developers experience a delay of only a few seconds.

As described above, the debugging user interface is updated with an intermediate slicing result twice per second. The slicing algorithm works its way backwards, beginning at the slicing criteria. Then, previous events are processed not ordered by their absolute position in the execution, but by their distance in the dependency graph. As a result, both the near past and long-running overarching method invocations are processed first.

This allows developers to begin debugging the slice almost immediately. From this point on, the slicer only needs to be faster than the developer moving backwards, which is generally given.

For interactive changes of slicing criteria, our experiments have shown that the incremental slicing algorithm is fast enough to not interrupt the developers flow. In particular, the most common operation – removing events by narrowing the slicing criteria – completes in less than a second even for large numbers of events. From this we conclude that our debugging approach, based on the Slice Navigator and on iterative slice refinements, is generally feasible.

VI. CONCLUSION

Many tools and approaches have been developed to help developers to find bugs in code. However, in practice few debugging tools are used that have features beyond symbolic debugging. We believe that one of the reasons for this is that developers don’t like to switch tools while they are in the process of hunting a bug. Short-term memory is limited and important insights are quickly forgotten if attention has to be divided to managing tools.

We presented the Slice Navigator, a UI component that directly integrates into the debugging workflow and uses a complementary tool, a dynamic slicer, to enrich the debugging experience. Slicing data is used to summarize relevant program state and allows to reach points of interest in the

execution history easier than with breakpoints. Iterative refinement of slicing criteria allows developers to “forget” about parts of the program that are not of interest.

Combined, these features allow for a new debugging workflow that allows to narrow down the cause of a fault efficiently by reducing the required amount of guesswork. An evaluation has shown that our algorithms perform fast enough to be usable in practice.

For future work, an improved semantic model for well defined interfaces (such as collections in the JDK) can be used to improve the quality of slices. For example, if an add-operation causes the internal array of an array list to be resized, all future gets will depend on that add, even though conceptually only one value was touched.

Furthermore, we wish to bring slicing to database applications. With the increasing relevance of “big data”, better debugging tools are needed for database applications. We expect that slicing can reduce not only the amount of code, but also the amount of data that needs to be analyzed to identify bugs in a program.

REFERENCES

- [1] B. Lewis, “Debugging backwards in time,” *Computing Research Repository*, vol. cs.SE/0310016, 2003.
- [2] M. Weiser, “Programmers use slices when debugging,” *Commun. ACM*, vol. 25, no. 7, pp. 446–452, 1982.
- [3] H. Agrawal and J. R. Horgan, “Dynamic program slicing,” in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI ’90. ACM, 1990, pp. 246–256.
- [4] A. Treffer and M. Uflacker, “Dynamic slicing with soot,” in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, ser. SOAP ’14. ACM, 2014, pp. 1–6.
- [5] M. Weiser, “Program slicing,” in *Proceedings of the 5th international conference on Software engineering*, ser. ICSE ’81. IEEE Press, 1981, pp. 439–449.
- [6] B. Korel and J. Laski, “Dynamic slicing of computer programs,” *Journal of Systems and Software*, vol. 13, no. 3, pp. 187–195, 1990.
- [7] R. M. Balzer, “EXDAMS: Extendable debugging and monitoring system,” in *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, ser. AFIPS ’69 (Spring). ACM, 1969, pp. 567–580.
- [8] H. Agrawal, R. A. Demillo, and E. H. Spafford, “Debugging with dynamic slicing and backtracking,” *Software: Practice and Experience*, vol. 23, no. 6, pp. 589–616, 1993.
- [9] S. I. Feldman and C. B. Brown, “IGOR: a system for program debugging via reversible execution,” in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, ser. PADD ’88. ACM, 1988, pp. 112–123.
- [10] H. Lieberman and C. Fry, “ZStep 95: A reversible, animated source code stepper,” 1995.
- [11] G. Pothier, r. Tanter, and J. Piquer, “Scalable omniscient debugging,” in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, ser. OOPSLA ’07. ACM, 2007, pp. 535–552.
- [12] A. Lienhard, T. Gırba, and O. Nierstrasz, “Practical object-oriented back-in-time debugging,” in *ECOOP 2008 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, J. Vitek, Ed. Springer Berlin Heidelberg, 2008, no. 5142, pp. 592–615.
- [13] M. Perscheid, M. Haupt, R. Hirschfeld, and H. Masuhara, “Test-driven fault navigation for debugging reproducible failures,” *Computer Software*, vol. 29, no. 3, pp. 188–211, 2012.