

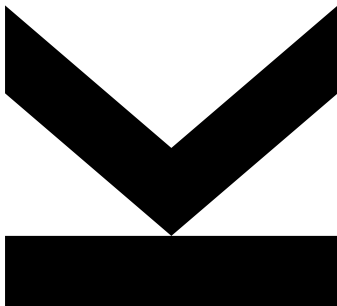
Author
Daniel Pfeffer, BSc

Submission
Institute for
System Software

Thesis Supervisor
Dipl.-Ing. Dr.
Markus Weninger, BSc

September 2026

Bypassing Debug Interfaces: A Language-Agnostic Graphical Debugger based on Code Instrumentation



Master's Thesis

to confer the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

Abstract

Graphical debuggers are valuable educational tools, requiring rich, fine-grained data (such as boolean expression results and array access patterns) to visualize program execution effectively. Existing standards such as the Debug Adapter Protocol (DAP) focus on coarse-grained control flow rather than fine-grained data tracing, while language-specific debug interfaces such as the Java Debug Interface (JDI) lack portability. We present a debugging architecture that bypasses such interfaces by leveraging Abstract Syntax Tree (AST) modification capabilities inherent to modern compilers (e.g., Roslyn, javac) to inject tracing statements directly into the program, turning it into a self-reporting artifact. These statements record a unified stream of micro-step events relevant to debug visualizations (such as variable mutations, conditional evaluations, and array accesses), transmitted over Protobuf-serialized Transmission Control Protocol (TCP) sockets. A language-agnostic state reconstructor then consumes this event stream to produce coherent call-stack and heap snapshots at each debugger step, fully decoupling visualization from any language-specific runtime. We use *language-agnostic* in a scoped sense: the protocol, the reconstructor, and the frontend carry no language-specific logic, and the claim is evaluated for statically typed, object-oriented languages. We demonstrate that this trace schema enables the existing graphical debugger *JavaWiz* (previously a Java-only tool reliant on JDI) to now support Java as well as C# with high fidelity without modifying the visualization frontend. Our evaluation shows that the trace-based architecture reduces the end-to-end stepping time of a debugging session to roughly one-third of that of the legacy JDI-based system, while the unified protocol amortizes the instrumentation cost across languages.

Kurzfassung

Grafische Debugger sind wertvolle Lehrmittel, die umfangreiche, detaillierte Daten (wie Ergebnisse von booleschen Ausdrücken und Zugriffsmuster auf Arrays) benötigen, um die Programmausführung effektiv zu visualisieren. Bestehende Standards wie das Debug Adapter Protocol (DAP) konzentrieren sich eher auf den groben Kontrollfluss als auf die detaillierte Datenverfolgung, während sprachspezifische Debug-Schnittstellen wie das Java Debug Interface (JDI) nicht portabel sind. Wir stellen eine Debugging-Architektur vor, die solche Schnittstellen umgeht, indem sie die in modernen Compilern (z. B. Roslyn, javac) vorhandenen Möglichkeiten zur Modifikation des Abstract Syntax Tree (AST) nutzt, um Tracing-Anweisungen direkt in das Programm einzufügen und es so zu einem selbst-berichtenden Artefakt zu machen. Diese Anweisungen zeichnen einen einheitlichen Strom von Mikro-Schritt-Ereignissen auf, die für Debug-Visualisierungen relevant sind (wie z. B. Variablenänderungen, Auswertungen von Bedingungen und Array-Zugriffe), und übertragen diese über Protobuf-serialisierte Transmission-Control-Protocol-Sockets (TCP-Sockets). Ein sprachunabhängiger Zustands-Rekonstruktor verarbeitet anschließend diesen Ereignisstrom und erzeugt kohärente Aufrufstapel- und Heap-Snapshots bei jedem Debugger-Schritt, wodurch die Visualisierung vollständig von jeder sprachspezifischen Laufzeitumgebung entkoppelt wird. Der Begriff *sprachunabhängig* ist dabei eng gefasst: Protokoll, Rekonstruktor und Frontend enthalten keine sprachspezifische Logik, und die Aussage wird für statisch typisierte, objektorientierte Sprachen belegt. Wir zeigen, dass dieses Trace-Schema es dem bestehenden grafischen Debugger *JavaWiz* (bisher ein rein Java-basiertes Tool, das auf JDI angewiesen war) ermöglicht, nun sowohl Java *als auch* C# mit hoher Genauigkeit zu unterstützen, ohne das Visualisierungs-Frontend zu ändern. Unsere Evaluation zeigt, dass die Trace-basierte Architektur die Gesamtdauer einer Debugging-Sitzung auf etwa ein Drittel jener des ursprünglichen JDI-basierten Systems reduziert, während das einheitliche Protokoll den Instrumentierungsaufwand über mehrere Sprachen hinweg amortisiert.

Master's Thesis

**A Trace-Based Graphical Debugger Based on
Execution Traces Collected via Source Code Instrumentation**

Dipl.-Ing. Dr. Markus Weninger, BSc

Institute for System Software

T +43-732-2468-4361

markus.weninger@jku.at

Student: Daniel Pfeffer, BSc

Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc

Start date: March 2025

The field of software debugging has seen significant advancements with the development of graphical debuggers, which provide visual representations of program execution to aid developers in identifying and resolving issues. One such tool is JavaWiz, a graphical debugger that currently collects data via the Java Debug Interface (JDI) during runtime. However, this approach has limitations, particularly in terms of language dependency and the overhead introduced during execution.

The proposed master's thesis aims to enhance JavaWiz by incorporating a new method of data collection through source code instrumentation. Instead of gathering execution data at runtime via JDI, additional code will be inserted into the program during compilation to emit execution traces. These traces will record various operations, such as method calls, variable accesses, and array manipulations, providing a detailed log of the program's behavior.

The primary challenge addressed by this thesis is the development of an instrumentation mechanism that can insert tracing methods into existing programs. This involves tracking local and field writes, reads, and maintaining information about currently live heap objects. Each object will be assigned a unique identifier, allowing the debugger to reconstruct the program state from the incremental execution traces.

The main objective of this thesis is to implement this new data collection method and integrate it with the existing JavaWiz debugger. The deliverables include:

- An instrumentation tool that inserts tracing code into programs.
- A mechanism to process the collected traces and reconstruct the program state.
- Modifications to the JavaWiz frontend to display the reconstructed state based on the new traces.
- A prototype for at least one language different from Java that is able to collect the same execution traces as the Java intrumenter.

The impact of this research is significant, as it aims to make JavaWiz language-independent. By demonstrating that source code instrumentation can provide all necessary debugging information without relying on a specific debugging interface, the tool can potentially support multiple programming languages, including Java, C#, Python, and Kotlin. This would greatly enhance the utility and applicability of JavaWiz in diverse development environments.

Students undertaking this thesis should have a strong background in compiler theory, experience with source code instrumentation, and proficiency in Java. They will develop skills in advanced debugging techniques, program analysis, and tool integration.

The expected outcome of this thesis is a publication that showcases the advantages of source code instrumentation for graphical debugging.

Modalities:

The progress of the project should be discussed at least every three to four weeks with the advisor. A time schedule and a milestone plan must be set up within the first two weeks and discussed with the advisor(s). It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis is expected to be finished before 28.02.2026.

Contents

Abstract	ii
Kurzfassung	iii
List of Tables	vii
List of Figures	viii
List of Listings	ix
1 Introduction	1
2 Motivation and Background	4
2.1 The Limits of Standard Protocols (DAP & LSP)	4
2.2 AST Modification Across Languages	5
2.3 The Graphical Debugger <i>JavaWiz</i>	6
3 System Architecture	8
3.1 Baseline Architecture	8
3.2 The “Trace-ify” Pipeline	9
3.3 Decoupling via Sockets	11
4 Execution Traces and AST Instrumentation	12
4.1 The Feature Mismatch Problem	12
4.2 The Protocol as an Intermediate Representation	14
4.3 Semantic Lowering	14
4.4 Fine-Grained Instrumentation Strategy	14
4.5 Instrumenting the AST	19
4.6 The I/O Proxy Problem	23
5 Collecting and Reconstructing Execution Traces	26
5.1 Event Collection and Chunking (TraceCollector)	27
5.2 Constructor Event Sanitization (ConstructorReconstructor)	28
5.3 The Shadow Memory Model (VirtualProgramState)	31
5.4 Simulating Debugger Control Flow (Step Handlers)	32
6 Evaluation	34
6.1 Functional Feasibility	35
6.2 Size of the Instrumentation Code	36
6.3 Performance Considerations	38
6.4 Threats to Validity	39
7 Limitations and Future Work	42
7.1 Limitations of Source Instrumentation	42
7.2 Future Work	43

Contents	vi
8 Related Work	45
9 Conclusion	47
Bibliography	48
Appendix A Supplementary Materials	54
A.1 Execution Trace Protocol: Protobuf Schema	54
A.2 Java Instrumentation Example	57
A.3 End-to-End Trace Example	58

List of Tables

4.1	Classification of Execution Trace Primitives.	13
4.2	Summary of AST instrumentation transformations. <i><Scope></i> denotes a generic placeholder for Local, Field, or Static contexts. . . .	19
6.1	Lines of code (LOC) of the legacy implementation, the two new instrumenters, and the shared, language-independent infrastructure.	37
6.2	Performance measurements for the iterative Fibonacci program. .	38

List of Figures

2.1	JavaWiz's <i>Array View</i> animates array operations, including indices written to and read from.	5
2.2	JavaWiz offers various educational visualizations, including <i>Flow Charts</i> as well as a <i>Memory Visualization</i>	6
3.1	The complete pipeline required to initiate a debugging session for Java: from receiving the initial code to instrumentation, running, and finally reconstruction and sending <i>TraceState</i> objects to the frontend.	10
3.2	Architecture Overview: Multiple language compilers feeding into a single, unified visualization server.	11
4.1	Overview of the instrumentation process for Java, showing how the original AST is transformed by the <i>TreeScanner</i> to modify existing nodes.	20
4.2	Overview of the instrumentation process for C#, showing how the original AST is transformed by the <i>SyntaxRewriter</i> to modify existing nodes.	22
5.1	Overview of the complete trace reconstruction pipeline, from the raw event stream emitted by the instrumented program to the discrete state snapshots consumed by the JavaWiz frontend. . . .	27
5.2	Constructor event sanitization for <i>A()</i> delegating to <i>A(int)</i> via <i>this(42)</i> (cf. Listing 5.1 and Listing 5.2). Because <i>this(42)</i> must be the first statement, the inner constructor's events (blue) are emitted before the outer constructor (orange) announces its entry. The <i>ConstructorReconstructor</i> buffers the inner sequence in <i>finishedParts</i> and re-appends it once the <i>NextLine</i> event marking the <i>this()</i> call is seen, restoring the nesting the frontend expects.	30
6.1	Visualization of an iterative Fibonacci implementation. We have already calculated the fifth Fibonacci number, 5 (<i>c</i>), and are preparing <i>a</i> and <i>b</i> for the sixth iteration.	34
6.2	Visualization of a singly linked list.	35
6.3	Visualization of a recursive descent parser. The parser has already consumed the first two positions (3 +), and is currently evaluating the third (5) in its fourth stack frame.	36

List of Listings

4.1	The original C# program. Its instrumented counterpart is shown in Listing 4.2.	15
4.2	Instrumented C#, showing injected line steps and variable tracking, based on Listing 4.1.	16
4.3	Handling of method invocation nodes in the Java AST.	21
4.4	How the C# instrumenter handles if statements to instrument the condition.	23
4.5	How the Java instrumenter handles if statements to instrument the condition.	24
4.6	Creation of I/O proxies.	24
5.1	Original class with a constructor A() delegating to A(int) via this(42).	29
5.2	Simplified instrumented pseudo code of Listing 5.1 (probe arguments abbreviated). Because this(42) must stay the first statement, A(int) emits all of its events (L13–L19) before A() announces its own entry (L5).	29
A.1	Proto3 schema for the execution trace protocol.	54
A.2	The original Java program, equivalent to Listing 4.1.	57
A.3	Instrumented Java, showing injected line steps and variable tracking, based on Listing A.2.	57
A.4	Raw event stream for one execution of Listing 4.1. Iterations 3–5 are elided.	58

Chapter 1

Introduction

Debuggers are arguably one of the most sophisticated yet underutilized tools in software development. Beller et al. [3] report that 71% of developers with at least three years of experience still rely on *printf debugging*, suggesting a misalignment between standard tooling and user needs.

While the debugging ecosystem has optimized for the inspection of program state, it has largely neglected the visualization of program flow and the novice-accessibility of debuggers. For this, we must distinguish between industry debugging (efficiency-focused) and educational debugging (comprehension-focused). Professionals use debuggers surgically to “teleport” to a crash state, minimizing mean time to repair. However, for novices (i.e., learners lacking a mental model), this teleportation is counterproductive. They first need to acquire skills to quickly traverse logic, loop iterations, and variable mutations to understand the system. For this demographic, the “noise” of execution, which standard tools skip, is actually the required signal.

Current graphical debuggers, i.e., visualization tools that attempt to ease the onboarding to debugging for novices, often rely on language-specific runtimes (e.g., the Java Debug Interface (JDI) or the .NET debugging API *ICorDebug*), creating “Language Silos”. Crafting visualizations for such debuggers is a labor-intensive process, and restricting these debuggers to a single source language by using a language-specific interface presents two systemic barriers:

- Standard debugging interfaces provide access to the “What” (state), but especially visualizations for novices require the “Why” (transitional logic). For instance, understanding index-based array accessing requires visualizing pointers (cf. Figure 2.1), which standard tools omit.
- Standard debugging interfaces are language-specific, i.e., they are tailored to debug one specific language, and the debugger generally has to be implemented in the language we intend to debug. While there are protocols that try to standardize debugging behavior, such as the Debug Adapter Protocol (DAP) [24], they lack fine-grained data extraction required for visualization. Consequently, without using a standardized protocol, porting a tool from, for example, Java to C# traditionally necessitates a complete rewrite of the debugging / extraction layer.

To resolve these challenges, we introduce a framework that shifts debugging logic from the *execution time* to the *compile time*. By injecting debugging probes directly into the target program using Abstract Syntax Tree (AST) modification, the program becomes a self-reporting artifact, eliminating the need for language-specific debug interfaces.

We refer to this architecture as *language-agnostic*, and we use the term in a deliberately scoped sense throughout this thesis. Language-agnosticism here is a property of the *architecture*, not a claim of universal language coverage: the trace protocol, the state reconstructor, and the visualization frontend contain no language-specific logic, and the instrumenter is the single component that must be provided per language. The scope of this claim is the family of statically typed, object-oriented languages that share the constructs our trace protocol models, namely classes, fields, methods, block scoping, and array access. Java and C# serve as our two representative implementations of this family. Languages outside it, in particular dynamically typed and functional languages, are beyond the scope of this thesis; we revisit this boundary in Section 6.4. The original task description envisions support for Java, C#, Python, and Kotlin. Kotlin appears compatible with the scoped family and would presumably require only an instrumenter, although we did not evaluate it; Python falls outside the family, and we defer it to Section 7.2.

To guide our evaluation, we formulate the following research questions:

- RQ1** Can the compiler-first approach correctly capture and reconstruct complex execution flows across different object-oriented programming languages? (Section 6.1)
- RQ2** To what extent does the language-agnostic architecture reduce the implementation effort when adding support for an additional language? (Section 6.2)
- RQ3** Does the resulting performance overhead permit interactive use in pedagogical debugging scenarios? (Section 6.3)

Thus, this thesis makes the following contributions:

1. We present a debugging architecture where the debuggee emits an execution trace via sockets by injecting tracing statements at compile-time. Instead of having to implement a complete debugger per target language, this architecture reduces the development effort to well-defined AST modifications to emit execution traces. This decouples the data collection process by not relying on language-specific debugging interfaces (Chapter 3).
2. We define a protocol that captures micro-steps (e.g., loop conditionals, variable assignments, heap object creation) to reconstruct execution flow rather than just state snapshots (Chapter 4).
3. We report a case study in which we modified the graphical debugger JavaWiz [48], a previously Java-only tool reliant on JDI, to now support Java and C# by leveraging our debugging architecture, without modifying its visualization frontend. We demonstrate that AST-based source-level instrumentation significantly reduces the engineering effort (Chapter 6).

The remainder of this thesis is structured as follows. Chapter 2 motivates the work by analyzing the limitations of standard debug protocols and introducing JavaWiz. Chapter 3 presents the compiler-first architecture and its socket-based transport. Chapter 4 defines the execution trace protocol and describes how the AST instrumenters emit it for Java and C#. Chapter 5 explains how the backend reconstructs coherent program states and simulates debugger stepping from the event stream. Chapter 6 answers the research questions in a case study and discusses threats to validity. Chapter 7 discusses limitations of the

approach and future work, Chapter 8 positions the thesis among related systems, and Chapter 9 concludes.

Chapter 2

Motivation and Background

To justify the architectural shift from debugging using established interfaces to compile-time instrumentation, we first analyze the technical constraints of the current ecosystem. This section examines the data granularity limitations of existing standard protocols (such as DAP), explores the cross-language potential of Abstract Syntax Tree (AST) modification, and introduces JavaWiz, the pedagogical tool that served as the primary motivation for this research.

2.1 The Limits of Standard Protocols (DAP & LSP)

The software tooling landscape has been significantly reshaped by the introduction of the Language Server Protocol (LSP) [25] and the Debug Adapter Protocol (DAP) [24]. While LSP focuses on providing static information to power features such as auto-complete and refactoring, DAP is designed to provide information about an application's state at run time, making the latter most relevant to our work.

By decoupling development tools (IDEs) from language runtimes and debuggers, respectively, these protocols successfully reduce the $N \times M$ complexity problem, i.e., the need for a dedicated integration between each of N development tools and each of M languages, for standard editing and debugging tasks. DAP, in particular, abstracts the heterogeneity of underlying debuggers (such as GDB, JDI, or the Python Debugger) into a uniform interface based on JSON Remote Procedure Call (JSON-RPC), allowing a single IDE to support stepping, breakpoints, and variable inspection across multiple languages without having in-depth knowledge of the underlying debuggers.

However, DAP was designed primarily to serve the needs of generic *stop-and-inspect* industry debuggers. The protocol dictates a coarse-grained interaction loop: the user halts execution at a statement, inspects the memory snapshot, and resumes. This abstraction layer, while beneficial for compatibility, actively hides the more fine-grained execution details required for pedagogical visualization.

Specifically, DAP lacks the fidelity to represent the intra-statement flow, operations that occur within a single line of code. We identify two critical deficiencies in DAP that prevent its use for detailed flow visualization within JavaWiz.

- Re-evaluating expressions for visualization purposes can lead to unintended side effects. For example, consider the condition `calcAndPrint() < 100`, where `calcAndPrint` performs a calculation and outputs a message.

Re-executing this condition, for example using DAP's `EvaluateRequest`, would result in duplicate outputs, i.e., unintended side effects. Similar problems arise if the function involves non-deterministic elements such as `Math.random()`.

- Visualizing interactions with data structures such as arrays requires not only information about *modifying* data, but also about *data reads*. Educational visualizations such as JavaWiz often animate these as distinct index movements, as shown in Figure 2.1, to explain array manipulation algorithms.

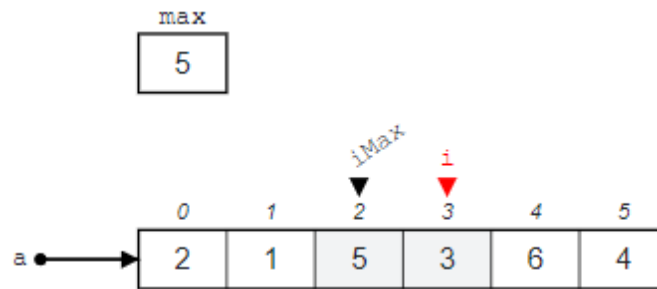


Figure 2.1: JavaWiz's Array View animates array operations, including indices written to and read from.

It is theoretically possible to extend DAP to support these features. However, doing so would require implementing custom protocol extensions for every target language adapter. This approach would effectively reintroduce the $N \times M$ complexity problem, defeating the very purpose of utilizing a standard protocol. Consequently, high-fidelity pedagogical tools require a more fine-grained data source.

2.2 AST Modification Across Languages

While execution environments exhibit heterogeneity, ranging from the stack-based architecture of the Java Virtual Machine (JVM) to the register-based Common Language Runtime (CLR) and the interpreter loops of CPython, the compilation process shares a fundamental commonality: prior to the generation of platform-specific machine code, bytecode, or interpretation, most modern compilers model the source program as an Abstract Syntax Tree (AST).

The AST serves as a canonical representation of logic, stripping away syntactic sugar while preserving the hierarchical flow of execution. Crucially, the structural concepts within an AST are largely similar across imperative languages. A `WhileLoop` node in a Java AST consists of a `Condition` and a `Body`, a structure nearly identical to that of a C# or Python AST.

Furthermore, the barrier to entry for AST manipulation has lowered significantly in recent years. Modern compiler infrastructures expose robust public APIs for syntax analysis and modification:

- Java: The Compiler Tree API (part of `javac`) enables inspection and modification of the AST during compilation, often invoked via Annotation Processors.

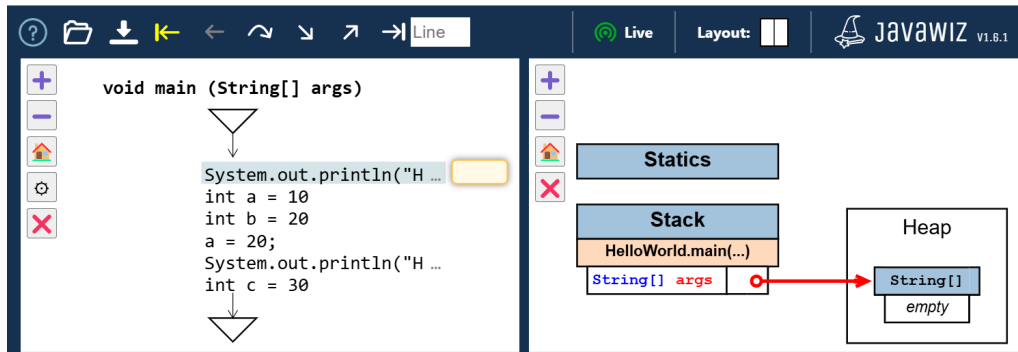


Figure 2.2: JavaWiz offers various educational visualizations, including *Flow Charts* as well as a *Memory Visualization*.

- **C#:** The Roslyn Compiler Platform (specifically the `SyntaxRewriter` class) provides an immutable, high-fidelity API that treats code analysis and transformation as a first-class citizen of the language ecosystem.
- **Python:** The standard library includes the `ast` module, which allows developers to parse source code into nodes, modify the tree structure, and compile it back into bytecode objects dynamically at run time.

By instrumenting at the AST level, we enforce unified behavior before the code ever reaches the divergent runtimes. Instead of writing one adapter for the JVM's event bus and another for the CLR's profiling API to detect a variable assignment, we inject a uniform `traceLocal()` probe directly into the source logic. Consequently, the instrumented program executes our tracing logic as if it were part of the original user program, effectively bypassing the need for specialized debugger interfaces entirely.

2.3 The Graphical Debugger *JavaWiz*

This research is motivated by JavaWiz, a graphical debugger designed for software engineering education [48]. JavaWiz supplements standard debugging features with educational visualization, as shown in Figure 2.2. Its features include (but are not limited to) *time travel navigation* (the ability to step backwards to re-examine past program states), an essential feature for in-class debugging presentations, a *flowchart* view that depicts the program and highlights its active elements, and a *memory* view that forms a graph-based representation of the stack and heap to explain reference semantics.

In its original implementation, JavaWiz relied on a hybrid architecture: it extracted the AST to perform modifications and the Java Debug Interface (JDI) for run-time control and data extraction [48]. While effective for standard Java environments, this architecture revealed scalability barriers, especially in the context of supporting multiple languages:

- The reliance on JDI, a heavily specialized JVM debug interface, meant that porting JavaWiz to C# or Python would require a complete rewrite of JavaWiz's debug environment (which we call the *backend*). This is mostly due to the fact that nearly all debugging interfaces for a given language

have to be used from within the same language. For example, to support C#, we would have to develop a C# JavaWiz backend, written in C#, that utilizes *ICorDebug API* to talk to C# programs, alongside a Java JavaWiz backend, written in Java, that utilizes JDI to talk to Java programs. Then, program states extracted using either the *ICorDebug API* or JDI would have to be transformed into the state schema that JavaWiz's frontend relies on.

- Capturing expressions (such as loop conditions) and array interactions required workarounds involving temporary variables and helper methods to trick JDI into reporting data it was not designed to expose. This already requires additional source code modification on top of JDI as a debugging interface.

These limitations motivated us to develop a language-agnostic debugging architecture to decouple JavaWiz's visualizations from the underlying runtime environment.

Chapter 3

System Architecture

To overcome the portability limitations of debug-interface-based debuggers, we propose a compiler-first architecture. Unlike traditional approaches that act as external observers, monitoring the debuggee process from the outside, our architecture transforms the user’s program into a self-reporting artifact. We organize the discussion as follows:

- Section 3.1 analyzes the baseline architecture of the original JavaWiz system.
- Section 3.2 introduces the “Trace-ify” pipeline, our proposed mechanism for shifting debugging logic from the runtime to the compiler via AST instrumentation.
- Section 3.3 describes our socket-based transport strategy, which eliminates the need for shared memory or foreign function interfaces, ensuring the system can operate in containerized or restricted environments.

3.1 Baseline Architecture

To understand the necessity of our proposed framework, we first describe the architecture of the original JavaWiz system [48]. As illustrated in the original JavaWiz paper [48], the baseline system is composed of three distinct parts: the frontend, the backend, and the debuggee.

The Frontend The frontend is a web-based visualization layer hosted within Visual Studio Code. It is responsible for rendering the specific educational views and is designed as a consumer of data. It receives a stream of serialized “Trace States” via WebSockets and renders them. Because it is decoupled from the execution logic, this component remains unchanged in our proposed language-agnostic architecture.

The Backend The backend acts as the central orchestrator, performing two tightly coupled roles:

1. Upon startup, the backend parses the user’s source code to generate an AST. The AST is used for two distinct tasks: (a) on the frontend (for example within the Flow Chart Visualization) and (b) to inject additional source code, necessary to extract fine-grained trace information, e.g., loop conditions.

2. Afterward, the backend acts as a wrapper around the language runtime. It is responsible for launching and attaching to the user's program (the *debuggee*), issuing control commands (e.g., `StepOver`, `StepInto`), extracting program state at step halts, and managing the mapping between the program states and JavaWiz's `TraceState` format.

The Debuggee The debuggee is the user's compiled Java program running in a separate JVM instance. In this baseline architecture, the connection between the backend and the debuggee is maintained via JDI.

This three-tier architecture creates a rigid dependency: the backend cannot function without the specific JVM primitives provided by the JDI. To support a new language (e.g., C#), the entire Debugger/Debuggee link, including the source code manipulation, would need to be re-implemented for (and in) the respective target language.

3.2 The “Trace-ify” Pipeline

To support multiple languages without implementing a language-specific debugger for each, we inverted the data extraction model. We shift the responsibility of retrieving dynamic program information from an external process using a debugging interface to the debuggee itself reporting this information as it runs. The resulting pipeline, illustrated in Figure 3.1, operates as follows:

- Dispatch (1): Upon receiving a user submission, i.e., source code (1a), the backend identifies the target languages and dispatches the source code to the corresponding language-specific instrumenter (1b).
- Instrumentation and AST Generation (2): The instrumenter first parses the source code into an *original AST* (2a), which is stored by JavaWiz for visualizations such as the flowchart diagram. Subsequently, the instrumenter again generates the AST, yet this time it injects probe calls into this *modified AST* to capture run-time behavior (2b) (further discussed in Chapter 4).
- Compilation (3): The modified AST is compiled into an executable artifact. Depending on the language, this results in either a native binary, a bytecode class file (e.g., Java), or an intermediate-language assembly (e.g., C#).
- Execution and Streaming (4): The backend executes the artifact (4a), establishing a dedicated I/O channel (via standard input) to send user input to the artifact. As the application executes, the injected probes emit atomic events (4b). For example, the statement `int a = 23` triggers a `local("a", "int", 23)` event, informing JavaWiz that the local variable `a` has been mutated.
- State Reconstruction (5): The backend consumes this stream of raw, language-agnostic events. A state reconstructor aggregates these atomic updates to build a coherent snapshot of the stack, the heap and all global (static) variables (further discussed in Chapter 5).

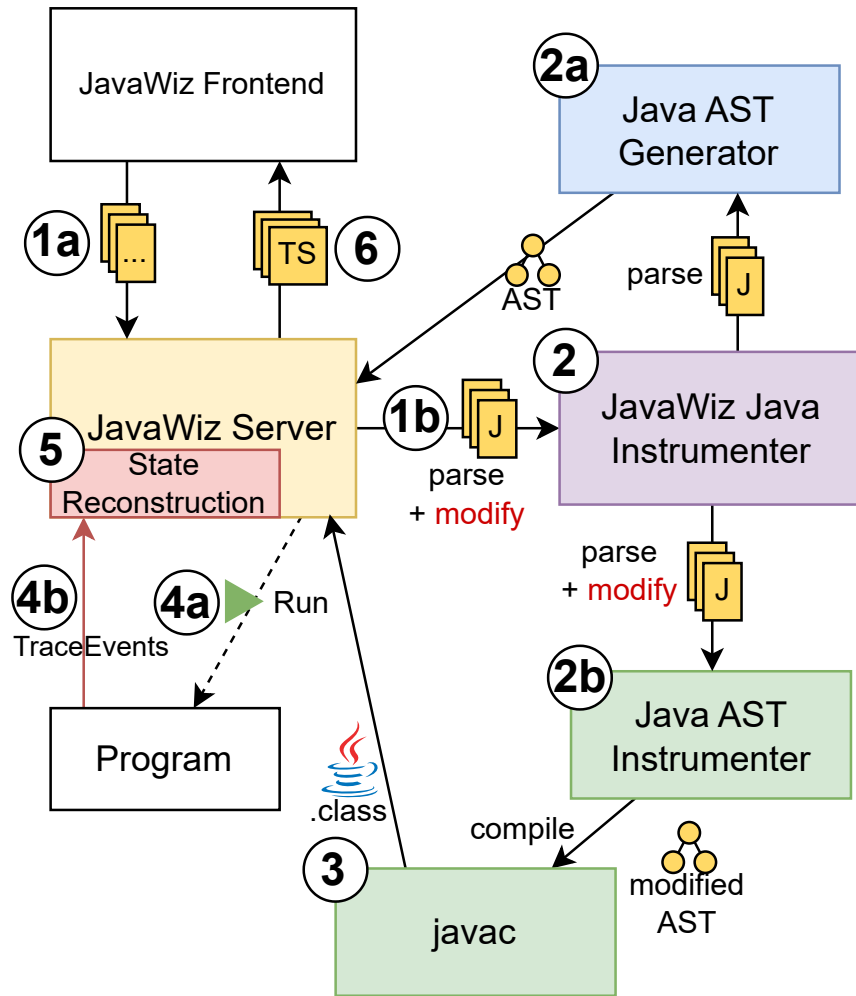


Figure 3.1: The complete pipeline required to initiate a debugging session for Java: from receiving the initial code to instrumentation, running, and finally reconstruction and sending *TraceState* objects to the frontend.

- **Visualization (6)**: Finally, the reconstructed state is converted into *TraceState* objects (i.e., JavaWiz’s existing data structure based on which visualizations are built upon), which are then pushed to the frontend to update and render the visualizations.

Figure 3.2 outlines JavaWiz’s new architecture, where the JavaWiz Server (i.e., the backend) is now able to handle Java *as well as* C# by having a C# Instrumenter, i.e., C# AST parsing and modifying capabilities.

Architectural consequences This design isolates the language dependency to a single component: the Instrumenter. To add support for a new language (for example, C#), a developer only needs to implement the logic to parse the C# AST and inject the probe calls. The complex logic of state reconstruction, stream management, and frontend visualization remains entirely generic and language-agnostic. The backend requires only trivial changes by defining how to invoke the instrumenter and how to run the resulting executable for a given

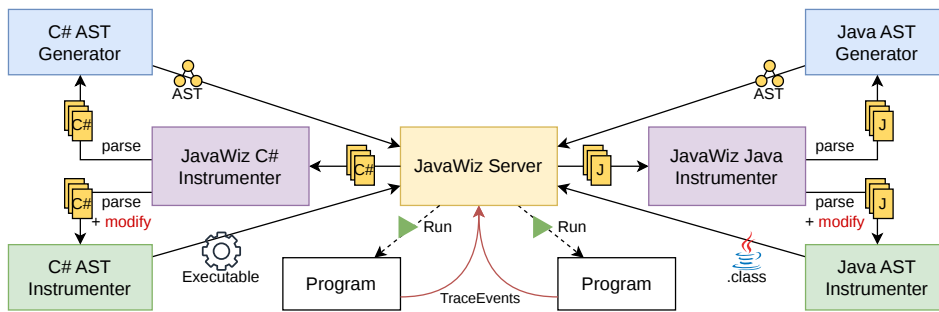


Figure 3.2: Architecture Overview: Multiple language compilers feeding into a single, unified visualization server.

language.

3.3 Decoupling via Sockets

To achieve language-independent transport within our scoped architecture, the communication mechanism between the debugger and the debuggee must be as portable as the instrumentation itself. While the legacy JavaWiz implementation relied on JDI, which enforces strict process coupling and JVM dependencies, our architecture introduces a socket-based telemetry layer. We utilize protocol buffers (*Protobuf*) over standard Transmission Control Protocol (TCP) sockets to stream execution events from the debuggee to the backend (Figure 3.1, (4b)). This design choice yields three critical advantages:

1. Traditional multi-language debuggers often rely on complex bridge technologies such as Java Native Interface (*JNI*) or Platform Invoke (*P/Invoke*) to access memory across runtime boundaries [14, 37, 44]. By treating the debuggee as a distinct network client, we eliminate these brittle dependencies. The instrumented artifact only needs to be capable of opening a socket and serializing a Protobuf message, capabilities present in the standard libraries of Java and C#, as well as of the other languages in the family considered here.
2. Protobuf provides a strictly typed, binary-efficient serialization format. This ensures that the high-frequency stream of trace events (e.g., variable mutations inside tight loops) incurs minimal serialization overhead compared to verbose text formats such as JSON [31].
3. The physical decoupling of the state reconstructor (backend) from the execution artifact (debuggee) allows them to operate in separate environments. For example, the backend can reside on a host server, while the instrumented artifact executes within a transient Docker container or a remote embedded device. This is particularly valuable for educational platforms that must sandbox student code to prevent malicious behavior. The debugger remains safely outside the sandbox, consuming telemetry over a controlled network port.

Chapter 4

Execution Traces and AST Instrumentation

While the “Trace-ify” pipeline resolves the difficulties of extracting execution data, it exposes a more fundamental theoretical challenge: semantic divergence across languages.

Although modern imperative languages share a common ancestry, typically characterized by block-structured control flow and object-oriented memory management, their semantics exhibit critical distinctions. A `struct` in C# possesses value semantics that do not exist in Java; a Python variable is a dynamic name binding rather than a fixed memory location; and C++ pointers introduce levels of indirection absent in managed runtimes.

To visualize heterogeneous concepts within a single, coherent interface, we cannot simply stream raw language constructs. Instead, we need a canonical representation of execution that operates at a level lower than the source code implies, and map language-specific behavior onto this ontology. This process mirrors the lowering phase in compiler design: high-level, language-specific constructs must be lowered into a generic Intermediate Representation (IR) without losing the intent of the original source code. Our unified model (see Table 4.1), represented as distinct trace events, acts as an IR between the highly language-specific source code and the `TraceState` objects that the `JavaWiz` visualization frontend expects (which were originally inspired by Java, more specifically JDI’s debugging data format).

In this chapter, we first analyze the friction between divergent language semantics and a unified visualization model: we define the feature mismatch problem inherent in cross-language debugging (Section 4.1), present the trace protocol as an intermediate representation (Section 4.2), and describe how language-specific features are lowered onto it, illustrated by mismatches between our Java and C# implementations (Section 4.3). Building on this trace format, the remainder of the chapter then describes how a program is rewritten to emit it: the instrumentation strategy (Section 4.4), its realization as an AST transformation (Section 4.5), and the handling of interactive input and output (Section 4.6).

4.1 The Feature Mismatch Problem

The primary challenge in designing a language-agnostic visualizer is that the visualization frontend is not truly agnostic in its semantic expectations. Originally architected for the Java ecosystem, the frontend expects a strict model

Category	Primitive(s)	Description
Control Flow & Scoping	MethodEntry / MethodExit	Defines the lifecycle of a stack frame.
	BlockEntry / BlockExit	Defines the visibility scope of local variables.
	NextLine	Marks the progression of the program counter as the currently next source code instruction line.
State Access & Mutation	LocalSet	Updates a variable within the current stack frame / scope.
	FieldSet / StaticSet	Updates a variable on the heap or global scope.
	ArraySet	Updates an indexed memory location.
	ArrayGet	Reports a read of an indexed memory location.
Memory Allocation	NewObject	Allocates a new heap entity.
	NewArray	Allocates a contiguous memory block, e.g., an array.
	NewString	Allocates a string value together with its backing character array.
Pedagogical Primitives	Print / Read	Abstracts I/O operations.
	Conditional	Explicitly reports the evaluation of boolean logic for visualizing branching.
Meta	RegisterClass	Announces a class and its source file when it is first loaded.

Table 4.1: Classification of Execution Trace Primitives.

based on the JVM memory architecture. This ontology has certain assumptions about its world:

- Firstly, execution occurs in strictly nested frames, which in turn contain local variables, which are popped upon method exit.
- All complex data structures are distinct memory entities (objects) referenced by unique identifiers and linked via pointers.
- Lastly, variables hold either primitive values or references to heap objects, with no ambiguity between the two.

When the target language conforms to this model (e.g., Kotlin), the mapping is straightforward. However, when languages have features that violate this ontology, such as C++'s pointer arithmetic, Python's dynamic attribute dictionaries, or C#'s struct value types, we encounter a feature mismatch.

4.2 The Protocol as an Intermediate Representation

Our trace protocol, presented in Table 4.1, defines a set of semantic events that represent the least common denominator of imperative execution, deliberately abstracting away runtime-specific implementation details. The protocol does not attempt to model every nuance of every language; instead, it currently provides a rather generic vocabulary.

The event set is designed to be additive. If a new visualization requires a concept not currently modeled (e.g., threading support), the protocol can be extended with new event types. Language instrumenters report which events they support, thus ignoring (or being unable to send) new event kinds just disables the respective visualizations. Without this capability negotiation, introducing new event types would necessitate a system-wide refactor of all supported languages.

4.3 Semantic Lowering

To support a new language, the language-specific instrumenter must perform a form of semantic lowering. It is the responsibility of the instrumenter to translate the divergent feature into its nearest equivalent in the unified event set.

This strategy prioritizes pedagogical usefulness over runtime truth. We aim to visualize the code in a way the learner understands it, not necessarily as the machine executes it.

A prime example is C#'s concept of properties. In C#, mutating `obj.Name` invokes a setter method (`set_Name()`) pushing a new stack frame and executing code. However, to a novice learner, this syntactically appears and behaves like a direct field access. Therefore, rather than visualizing the confusing internal method call, our C# instrumenter lowers this operation to a `FieldSet` event unless there is a custom setter or getter defined; then we trace the specific method call in detail. Similar “simplifications” were already present in JavaWiz's original version. For example, calls to a Java class's standard constructor, i.e., a constructor that could not be directly mapped to a source location, were not visualized in detail already. This deliberate simplification aligns the visualization with the conceptual model of the learner, masking the implementation details of the runtime.

4.4 Fine-Grained Instrumentation Strategy

The core of our contribution lies in the transformation of the user's source code into a self-reporting artifact based on the execution trace format presented above. This process must balance two competing requirements: fidelity, i.e., accurately representing the program's state, and transparency, i.e., ensuring the instrumentation does not alter the program's semantics.

To achieve balance between fidelity and transparency, we deliberately reject bytecode manipulation (e.g., Java Agent API, MSIL injection) in favor of source-level AST transformation. While such techniques are powerful, they operate at a level of abstraction that is too low for pedagogical visualization. Compilers often introduce synthetic artifacts, such as bridge methods, iterator state machines, or auto-boxing routines, that possess no direct correlation in the source code. Visualizing these artifacts would create cognitive noise for the learner. By operating on the AST, i.e., before bytecode is generated, we ensure that we only visualize concepts that explicitly exist in the student’s mental model.

Our instrumentation strategy targets three distinct dimensions of program execution: (1) *Control Flow*, (2) *State Mutation*, and (3) *Decision Logic*. These injected instrumentation statements are mostly wrapper functions of our more generalized protocol definition, defined in Section 4.2. They are used to simplify instrumentation and implementation of the creation of the necessary protocol objects to be sent to the debugging backend. To present their use, we iteratively show the transformation of the source code illustrated in Listing 4.1 to finally result in the transformed source code Listing 4.2.

```
1 class Example
2 {
3     public static void Main()
4     {
5         var array = new int[5];
6         for (int i = 1; i <= 5; i++)
7         {
8             array[i - 1] = i * i;
9         }
10    }
11 }
```

Listing 4.1: The original C# program. Its instrumented counterpart is shown in Listing 4.2.

Control Flow (The “Where”) To reconstruct the program’s execution context, we must track more than just the current line number. We must visualize the creation and destruction of stack frames and variable scopes. Therefore, our instrumenter injects explicit markers into the AST:

- To get notified when a new line begins in the original source code, we inject a `Recorder.NewLine(int line_number)` call before every executable statement (cf. Listing 4.2 L9, L12, L20, L24). This “heartbeat” of execution allows the frontend to highlight the active line and serves as the anchor point for `Step Into` requests.
- To visualize the call stack, we insert `Recorder.EnterMethod(name, args...)` at the start of every function body (cf. Listing 4.2 L5–L6) and `Recorder.ExitMethod()` in a finally block to ensure we capture every return point, including implicit returns and potential thrown exceptions (cf. Listing 4.2 L26–L30). The original method block is wrapped by the accompanying `try` (cf. Listing 4.2 L7–L25). Crucially, after the entry event,

```
1 class Example
2 {
3     public static void Main()
4     {
5         Recorder.EnterMethod(typeof(Example),
6             "Main", "Main()", typeof(void), 2);
7         try
8         {
9             Recorder.NextLine(4);
10            var array = Recorder.SetLocal("array",
11                Recorder.NewArray(new int[5]));
12            Recorder.NextLine(5);
13            Recorder.EnterBlock();
14            for (
15                int i = Recorder.SetLocal("i", 1);
16                Recorder.Condition(0, i <= 5, "i<=5");
17                Recorder.SetUnaryLocal("i", i++, i)
18            )
19            {
20                Recorder.NextLine(7);
21                Recorder.SetArray(array, i - 1, i * i, 0);
22            }
23            Recorder.ExitBlock();
24            Recorder.NextLine(9);
25        }
26        finally
27        {
28            Recorder.ExitMethod(typeof(Example),
29                "Main", 0, true);
30        }
31    }
32
33    static Example()
34    {
35        Recorder.RegisterType(typeof(Example),
36            "Example.cs");
37    }
38 }
```

Listing 4.2: Instrumented C#, showing injected line steps and variable tracking, based on Listing 4.1.

we also generate `SetLocal` events for every parameter to capture the initial values of parameters, establishing the baseline state of the new stack frame.

- Local variables in many languages (e.g., C#, Java, C++) are scoped to specific blocks (e.g., as part of loops or if-statements). To visualize variable shadowing and scoping in general, we wrap block bodies with `Recorder.EnterBlock()` (cf. Listing 4.2 L13) and `Recorder.ExitBlock()` (cf. Listing 4.2 L23).
- Finally, to visualize static fields, the instrumenter registers every class with the backend by injecting a `Recorder.RegisterType(className)` call. In languages such as Java or C#, this instrumentation is placed inside the static initializer (`<clinit>`) to ensure the class is reported the moment it is first accessed by the runtime (cf. Listing 4.2 L33–L37).

State Mutation (The “What”) Capturing variable updates requires intercepting assignments. A naive approach might insert a reporting call after each assignment (e.g., `x = 5; Report(x);`). However, this fails to capture the exact moment of mutation, particularly in nested expressions. Additionally, in these nested expressions, we would have to keep track of every variable that gets modified in all subexpressions, e.g., `x = (y += 1) + 5` would require multiple captures, one for each mutation (e.g., `x = (y += 1) + 5; Report(x); Report(y);`). Instead, we leverage assignment expressions, a language construct that most C-family languages (Java, C#, C++) possess. Therefore, using assignment expressions, a valid instrumentation of, e.g., `x = 5` becomes `Recorder.SetLocal("x", x = 5)`. Similarly, nested expressions, such as `x = (y += 1) + 5` becomes `Recorder.SetLocal("x", x = Recorder.SetLocal("y", y += 1) + 5)`. However, using our recorder in this fashion is only possible if the recorder itself behaves like an assignment expression, meaning by default the recorder methods responsible for state mutation should always return the value that the variable to trace receives. This way, the value is updated as in the original program flow while simultaneously pushing the new state to the event stream. This also simplifies working with the AST, as now, for an expression, we can transform the node recursively and place the recorder call with the transformed node as its argument in place of the original node.

Alternatively, if such language constructs are not possible, e.g., declaration with initialization, we devised the alternative to split the assignment accordingly: `int x = 5` becomes `int x = Recorder.SetLocal("x", 5)` (cf. Listing 4.2 L10–L11). This still preserves the exact moment of mutation even in complex expressions while allowing for more generalized assignments. Note that augmented assignments have to be handled in a similar fashion, e.g., `x += 5` becomes `x = Recorder.SetLocal("x", x + 5)`.

Unary operations such as `x++` or `++x` also have to be handled specifically, since simply transforming them into the form `x += 1` would alter program semantics. Thus, we introduced the wrapper `Recorder.SetUnaryLocal("x", x++, x)` to accurately return the correct value of the unary operation (second argument), in this case post-increment, but also to trace the correct value (third argument) the variable gets after the unary operation (cf. Listing 4.2 L17).

In addition to declarations and assignments, we have to ensure that we track new heap elements, i.e., objects and arrays, to correctly reference and visualize them. In our Recorder wrapper, this is done by either `Recorder.NewArray(<array construction>)` for arrays (cf. Listing 4.2 L11) or `Recorder.NewObject(<constructor call>)` for objects; both return the freshly created instance so that they can be nested inside the surrounding `Set<Scope>` call.

Branching and Logic (The “Why”) Standard debuggers often obscure the evaluation of conditional logic, jumping instantaneously from an `if` statement to the target block. To visualize the decision-making process, we wrap boolean predicates in a pass-through identity function. This identity function is invoked by including a unique condition identifier, the condition expression, and the string representation of the condition. For example, `if (x > 0)` becomes `if (Recorder.Condition(0, x > 0, "x > 0"))` (cf. Listing 4.2 L16). This transformation forces the runtime to report the result of the evaluation before the control flow diverges. This is critical for visualizing and teaching control flow concepts.

Mutating Arrays Mutating arrays has to be handled separately, as we want to capture (1) the array itself, (2) the index of the mutation, (3) and also the value the new array element receives. A naive implementation, e.g., `Recorder.SetArray(arr, index, arr[index] = 5)`, would not behave transparently when the index expression exhibits side effects, e.g., `arr[i++] = 5` would become `Recorder.SetArray(arr, i++, arr[i++] = 5)` and thus the index would be incremented twice. To address this behavior, we decided on handling the assignment of the array within our instrumentation method and invoking the instrumentation with a special instruction to reconstruct the original expression. For example, `arr[i++] = 5` becomes `Recorder.SetArray(arr, Recorder.SetUnaryLocal("i", i++, i), 5, 0)`, where inside `SetArray` the given value is assigned to `arr` at the given index (cf. Listing 4.2 L21). As augmented assignments are also possible for arrays, we have to account for that; therefore, we pass an instruction to the method, e.g., whether the value should be simply assigned (for this implementation, 0), or whether the value should be added to the current value of the array at this index (for this implementation, 1).

Reading Arrays As motivated in Section 2.1, visualizing array algorithms requires knowledge of *read* accesses as well as writes. Array element reads are therefore wrapped in `Recorder.GetArray(arr, index)`, which emits an `ArrayGet` event for the array and index and returns the element unchanged. For example, `x = arr[i]` becomes `x = Recorder.SetLocal("x", Recorder.GetArray(arr, i))`, allowing the frontend to animate the read index before the written variable is updated.

Therefore, in total, our instrumentation strategy is defined by the transformations summarized in Table 4.2:

Category	Original Construct	Transformation Strategy & Injected Probes
Control Flow	Method / Constructor	Wrap body in try-finally. Inject EnterMethod() at entry, ExitMethod() in finally. Parameters captured via SetLocal().
	Block Scope	Inject EnterBlock() and ExitBlock() at scope boundaries.
	Line Step	Prepend NextLine(line_id) before every statement.
	Conditional Logic	Wrap condition in Condition(id, expr, "expr_str") to record truth value.
State Mutation	Variable Declaration	Wrap initialization in Set<Scope>(...). <i>Note: Field/Static initializers are hoisted to their respective constructors.</i>
	Variable Assignment	Wrap assignment expression in Set<Scope>(...).
	Unary Operations	Wrap in SetUnary<Scope>(...) to capture pre- and post-mutation states.
Memory & Data	Object / Array Creation	Wrap instantiation in NewObject(...) or NewArray(...).
	Array Access & Mutation	Replace direct access with GetArray(...) and SetArray(...).
Meta	Static Type Registration	Inject RegisterType(...) during class initialization.

Table 4.2: Summary of AST instrumentation transformations. <Scope> denotes a generic placeholder for Local, Field, or Static contexts.

4.5 Instrumenting the AST

Our general instrumentation strategy is implemented by traversing the AST and applying the transformations described in Section 4.4. However, the specific implementation details differ depending on the language and its underlying compiler infrastructure.

In Java, we utilize the *Compiler Tree API* by implementing a custom *TreeScanner* that visits each node in the AST. A defining characteristic of this API is that transformations are applied mutably during compilation. Rather than generating a completely new AST and swapping it with the old one, we can modify existing nodes in place.

However, the *Compiler Tree API* is inherently designed for top-down traversal. To successfully mutate the tree, we must ensure that all children of a given node are processed and modified before invoking the `super.visit...()` method on the parent node. If the parent is visited first, the transformations applied to its children will not be persisted in the final AST. This architectural lim-

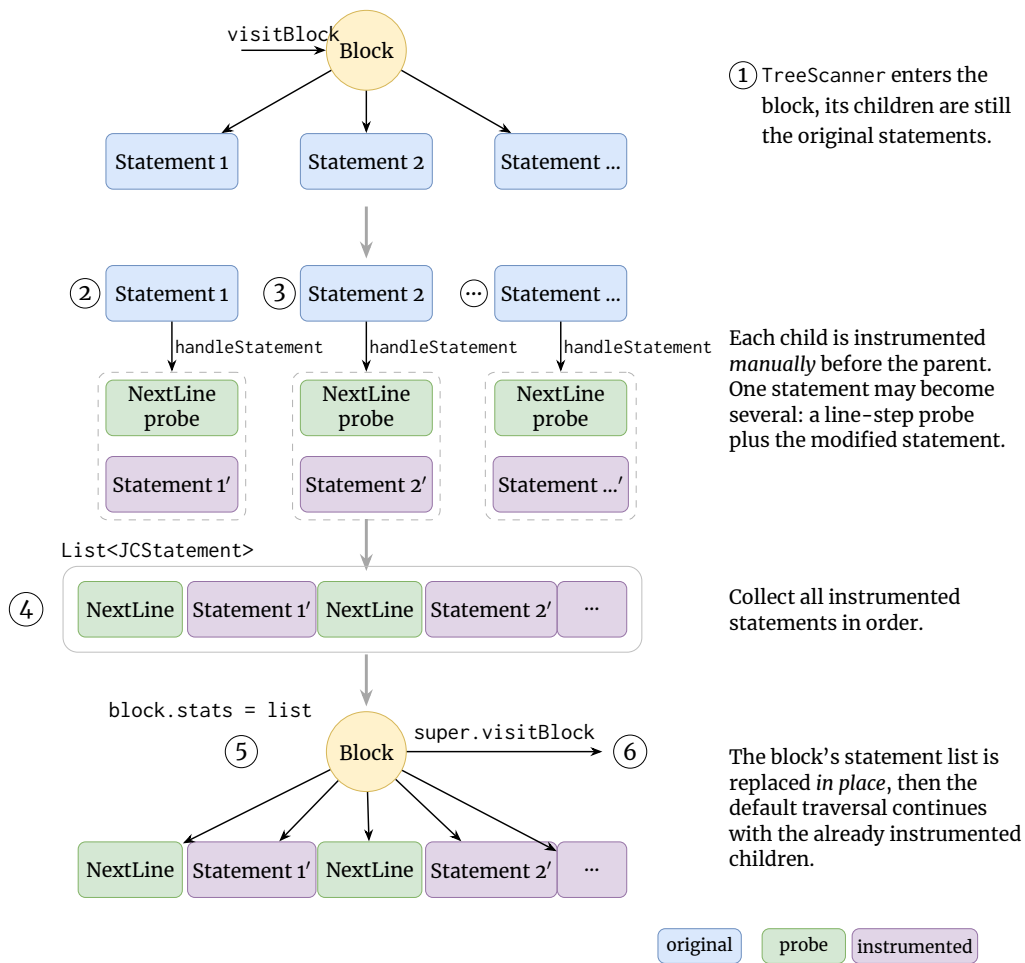


Figure 4.1: Overview of the instrumentation process for Java, showing how the original AST is transformed by the TreeScanner to modify existing nodes.

itation means we cannot rely solely on the default recursive behavior of the TreeScanner. Instead, we must manually orchestrate the traversal and instrumentation of child nodes. This presents a specific challenge for nested expressions and control-flow constructs containing block bodies (e.g., `if` statements, method declarations, or local blocks).

For example, even though a block body itself is not in our list of structural transformations, we must explicitly override the visit to a `JCBlock` node to ensure the statements within it are instrumented. As illustrated in Figure 4.1, when visiting a `JCBlock` node ①, we manually trigger the instrumentation of its child statements ②-③. We then collect these modified children ④, update the block body in place ⑤, and finally invoke `super.visitBlock()` with the updated node ⑥.

This manual delegation extends beyond block statements to all AST nodes containing nested elements. For instance, while a method invocation might not inherently require modification, its arguments could contain expressions that do

(e.g., an inline assignment). Listing 4.3 demonstrates how we handle method invocations within the overarching `handleExpression` routine, which is called for every expression node. First, we verify that the call is not targeting our own Recorder methods to prevent infinite instrumentation loops ①. Next, we recursively instrument the method's arguments ②. We then create a new method invocation node containing these modified arguments ③ and return it ④. This returned node is passed back up the call stack, allowing the parent node to invoke its respective `super.visit...()` method and persist the changes into the generated bytecode. Consequently, a nested expression such as `foo(i++)` is correctly transformed into `foo(Recorder.SetUnaryLocal("i", i++, i))`.

Ultimately, this mutable approach presents distinct trade-offs. The primary advantage is efficiency: it requires only a single pass over the AST and avoids the overhead of rebuilding the entire tree. It also seamlessly accommodates one-to-many node transformations, such as injecting a step-event probe immediately before a target statement. Conversely, the major disadvantage is the extensive manual bookkeeping required. Developers must meticulously track and explicitly traverse all children of a node, making the implementation less intuitive and more error-prone than a standard recursive approach. Failing to explicitly instruct a parent node to traverse its children risks silently skipping the instrumentation of all nested elements.

```

1 private fun handleExpression(
2     expr: JCTree.JCExpression
3 ): JCTree.JCExpression {
4     return when (expr) {
5         is JCTree.JCMethodInvocation -> {
6             // (1)
7             if (expr.isTracingCall()) {
8                 return expr
9             }
10            // (2)
11            val modifiedArgs = handleFnArgs(expr.args)
12
13            // (3)
14            val modifiedExpr = treeMaker.at(expr.pos).Apply(
15                expr.typeargs,
16                expr.meth,
17                List.from(modifiedArgs)
18            ) as JCTree.JCMethodInvocation
19
20            // (4)
21            modifiedExpr
22        }
23    }

```

Listing 4.3: Handling of method invocation nodes in the Java AST.

For C#, we utilize the *Roslyn Compiler Platform*, which provides a more intuitive API for AST manipulation. Unlike Java's mutable approach, the Roslyn API treats the AST as strictly immutable. We cannot modify existing nodes in place;

instead, we must replace them by returning newly constructed, instrumented nodes from the respective `Visit...()` methods.

This immutability enforces a bottom-up transformation strategy. We only need to intercept the AST traversal at nodes that explicitly require instrumentation (e.g., an assignment expression), while relying on the recursive base class to handle the children automatically. For instance, when processing a `Block` node in C#, we do not need to manually orchestrate the transformation of its contents. As illustrated in Figure 4.2, when a `Block` node is processed by the *base implementation* of the `SyntaxRewriter` (①), the visitor automatically invokes the corresponding methods for all child statements, such as `VisitStatement()` (②). We simply override the visit method for the specific statement type we wish to instrument (③) and return the modified node, which naturally propagates back up the call stack (④). This automated traversal is seamlessly applied to all children of the original `Block` node (⑤).

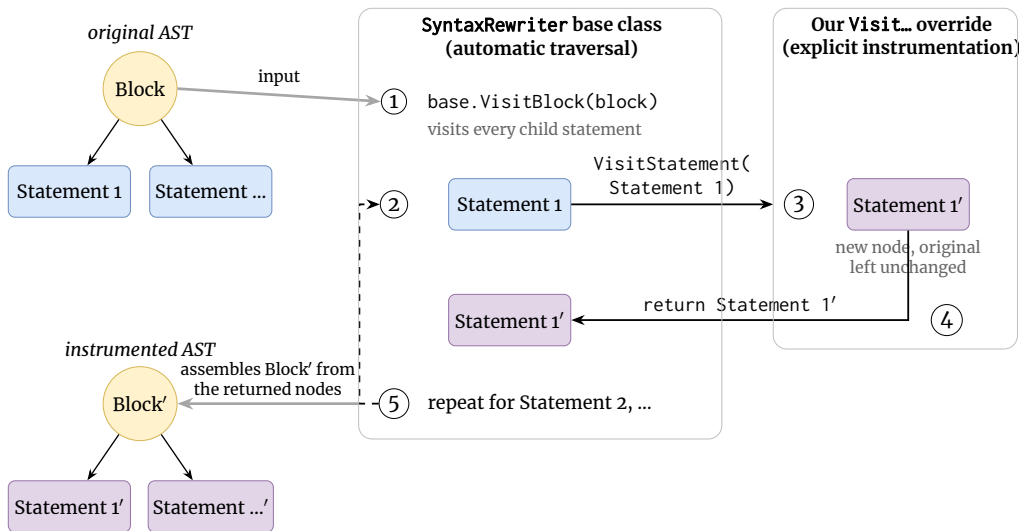


Figure 4.2: Overview of the instrumentation process for C#, showing how the original AST is transformed by the `SyntaxRewriter` to modify existing nodes.

Consequently, this approach heavily simplifies implementation. In Java, we must manually unpack a method invocation to ensure its arguments are instrumented. In C#, this manual unpacking is unnecessary; the arguments are automatically visited and instrumented by the recursive AST traversal. This drastically reduces the risk of accidentally omitting required instrumentation, as developers only need to handle nodes requiring explicit modification.

The contrast between the two paradigms is especially clear when handling `if` statements. In C# (Listing 4.4), we simply override the `VisitIfStatement` method of the `SyntaxRewriter` class. First, we invoke the base class visitor (①) to ensure the child nodes (e.g., the `then` and `else` blocks) are traversed and modified automatically. Once the base visit completes, we copy the `IfStatementSyntax` node containing these newly modified children (②). Finally, we construct and return a new `IfStatementSyntax` node where the condition is wrapped in our instrumentation method via

SourceGenerator.CreateCondition (2).

Conversely, in Java (Listing 4.5), we must manually intercept the JCIf node wherever a JCStatement can occur by calling our custom handleStatement method (1). For a JCIf node, we must explicitly instantiate a new JCIf object (2) containing the instrumented condition (3). We also manually create explicit block statements for the then and else branches (4) (which are then processed by the TreeScanner similarly to method declarations). Furthermore, because a single statement in Java might be transformed into multiple statements (e.g., injecting a line-step probe before the modified statement, as seen in Figure 4.1), we cannot simply return the new node. Instead, we must append the modified JCIf node to a running list of statements (5).

While the Roslyn API significantly streamlines the instrumentation logic by eliminating the need to track unmodified parent nodes, it does introduce a notable drawback. Because the AST is immutable, instantiating a new node strips it of its original line and column source span information. To preserve this crucial debugging data, we are forced to execute two separate transformer passes over the entire AST: the first pass annotates every statement with explicit line information, and the second pass injects the actual debugging instrumentation.

```

1 public override SyntaxNode VisitIfStatement(
2     IfStatementSyntax node)
3 {
4     // (1)
5     var based = base.VisitIfStatement(node) as IfStatementSyntax;
6
7     // (2)
8     return based.WithCondition(
9         SourceGenerator.CreateCondition(based.Condition));
10 }
```

Listing 4.4: How the C# instrumenter handles if statements to instrument the condition.

4.6 The I/O Proxy Problem

A specific challenge arises when visualizing interactive programs that require user input (e.g., programs using Scanner in Java or using Console in C#). These operations block the execution thread until data is returned from stdin. In our architecture, this blocking behavior creates an ambiguity. If the stream of trace events suddenly ceases, the visualization engine cannot distinguish between three potential states:

1. Processing: The debuggee is performing a long-running calculation.
2. Blocked: The debuggee is waiting for user input.
3. Terminated: The debuggee has crashed or hung silently.

```

1 private fun handleStatement( // (1)
2   stat: JCTree.JCStatement?
3 ): List<JCTree.JCStatement> {
4   if (stat == null) {
5     return List.nil()
6   }
7   val newStats = ListBuffer<JCTree.JCStatement>()
8   when (stat) {
9     is JCTree.JCIf -> {
10      // (2)
11      val newCondition = treeMaker.If(
12        generator.generateTraceRecordCondition(stat.cond), // (3)
13        treeMaker.Block( // (4)
14          0,
15          List.of(stat.thenpart)
16        ),
17        stat.elsepart?.let {
18          treeMaker.Block( // (4)
19            0,
20            List.of(it)
21          )
22        }
23      )
24      newStats.append(newCondition) // (5)
25    }
26    ...

```

Listing 4.5: How the Java instrumenter handles if statements to instrument the condition.

```

1 static Example()
2 {
3   Recorder.RegisterType(typeof(Example), "Example.cs");
4   Console.SetIn(new TracingInputStream(Console.In));
5   Console.SetOut(new TracingPrintStream(Console.Out));
6 }

```

Listing 4.6: Creation of I/O proxies.

The legacy JavaWiz architecture attempted to resolve the ambiguity through probabilistic heuristics. Upon issuing a “Step” command, the backend would start a timer (of 1,000 ms). If no events were received within this window, the system would assume the debuggee was blocked on I/O and waiting for user input (since (1) JDI would report debuggee crashes and (2) such long-running calculations without intermediate steps were assumed to be uncommon). While this approach mostly worked “well enough”, it was flawed due to its reliance on wall-clock time. On slower hardware or under heavy load, especially calls to more complex third-party functions (where no intermediate steps are reported) might exceed the timeout, causing the frontend to erroneously ask for input while the program was actually still running and not requiring user input.

Our proposed architecture abandons heuristics in favor of explicit I/O stream proxying. Instead of accessing the original streams directly, the instrumenter injects wrappers around the standard stream facilities (e.g., replacing `System.in` in Java or wrapping `Console.In` in C#, as shown in Listing 4.6, L4 and L5). This proxy enforces a notify-then-block approach:

1. Firstly, when the user code invokes a read operation, the call is intercepted by the proxy.
2. Afterward and before invoking the underlying OS primitive, the proxy emits a specific Read event to the trace stream. This explicitly informs the back-end state machine to transition to an “Awaiting Input” state, which in turn informs the user on the frontend to input data.
3. Only after the event is flushed to the socket does the proxy call the actual blocking read (e.g., `proxied.Read()`).
4. Finally, once data is received, this data is sent to the trace, the proxy returns the value to the user code, and the normal trace stream resumes.

Output is proxied in the same manner: the wrapped output stream (`TracingPrintStream`, Listing 4.6 L5) forwards each write to the original stream and additionally emits a `Print` event carrying the written text, so that the frontend can display program output in step with the visualization instead of reading it from the process’s standard output.

By wrapping the I/O boundary, we convert an ambiguous silence in the trace stream into a distinct, meaningful state. This ensures that the frontend prompts the user for input only when the program is genuinely requesting it, regardless of execution speed or environment latency.

Chapter 5

Collecting and Reconstructing Execution Traces

While the instrumentation layer successfully emits a continuous stream of fine-grained execution events, this raw stream is semantically flat. A graphical frontend, such as JavaWiz, requires cohesive, discrete snapshots of the program's state bounded by logical user actions (e.g., executing a single line). This chapter describes the steps—collection, sanitization, state simulation, and flow control—that are necessary to turn the raw event stream into the `TraceState` objects that the JavaWiz frontend expects.

To bridge the gap between the continuous stream of fine-grained events and the discrete states required by the visualization UI, we designed an event-driven, multi-stage data processing pipeline, illustrated in Figure 5.1. This pipeline manages two distinct flows: the asynchronous push of data from the running instrumented program and the synchronous pull of state requested by the user.

The process begins with the *Instrumented Program* (1), which executes asynchronously and streams a sequence of *Continuous Raw Events* (A) over a socket connection. The *TraceCollector* (2) acts as the primary consumer of this event stream, further discussed in Section 5.1. For the majority of operations, it forwards events directly (B) to the *EventQueue* (4). However, because object-oriented initialization sequences often produce out-of-order execution traces, *Constructor Events* (C) are routed to the *ConstructorReconstructor* (3), further discussed in Section 5.2, which buffers and sanitizes them before forwarding the *Sanitized Events* (D) to the event queue.

The actual consumption of these queued events is entirely user-driven. When the *JavaWiz Frontend* (5) registers a *User Action* (E) (e.g., Step Into, Step Over), the request is dispatched to the *Step Handlers* (6), further discussed in Section 5.4. To fulfill the user's request, the active *Step Handler* triggers the *apply()* (F) routine on the *TraceCollector* (2).

During each invocation of the *apply()* routine, the *TraceCollector* sequentially *Consumes* (G) events from the queue (4). For each event consumed, it executes an *Apply Event* (H) operation on the *VirtualProgramState* (7), further discussed in Section 5.3, which simulates the program's memory and call stack.

The *TraceCollector* continues to consume and apply events until the *Step Handler* determines that a logical execution boundary has been reached (e.g., the completion of a source code line). At this point, the *VirtualProgramState* (7) *Yields*

State ①, producing a complete, static snapshot of the program’s simulated memory and control flow. The *Step Handlers* then package this snapshot into a *TraceState* and *Send TraceState* ⑪ back to the *JavaWiz Frontend* ⑤, allowing the UI to render the updated graphical visualization.

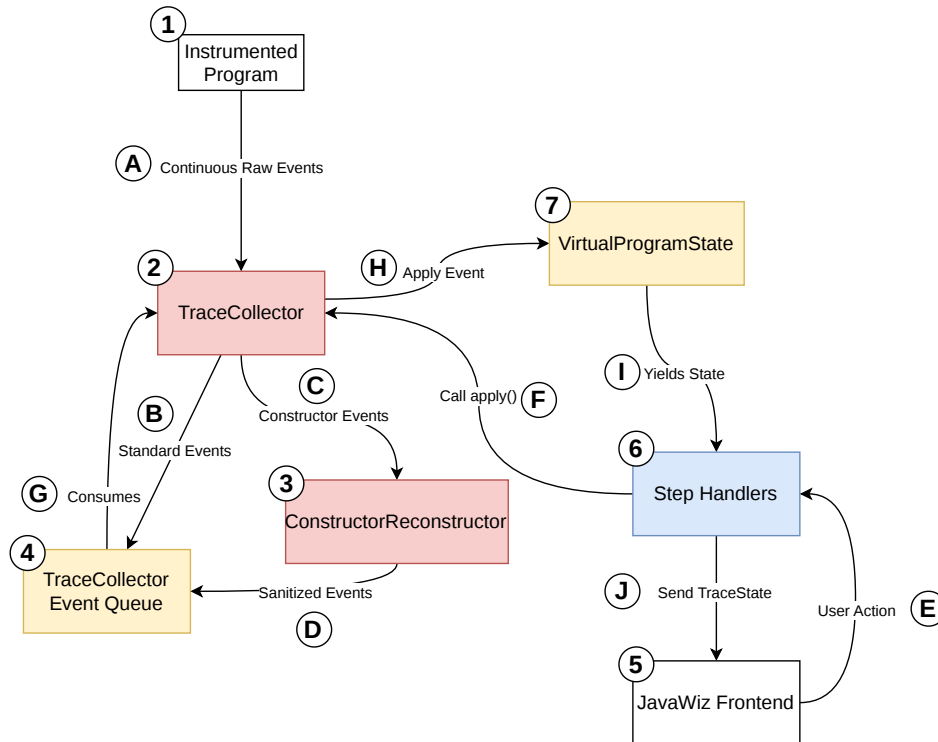


Figure 5.1: Overview of the complete trace reconstruction pipeline, from the raw event stream emitted by the instrumented program to the discrete state snapshots consumed by the JavaWiz frontend.

5.1 Event Collection and Chunking (TraceCollector)

The continuous stream of fine-grained events arriving from the instrumented program is semantically flat, meaning it lacks the concepts of a debugger step. To map this continuous stream to the discrete, line-by-line mental model expected by a user, the system must aggregate these events into logical execution chunks. This aggregation is managed by the *TraceCollector*.

Operating on a queue of pre-sanitized events (where structural anomalies have already been resolved), the *TraceCollector* functions as a stateful iterator. When the frontend requests the next program state, the collector invokes its core *apply()* routine. This routine pulls events from the queue sequentially and immediately dispatches them to a simulated memory model (the *VirtualProgramState*, details in Section 5.3) via the *applyTraceToState* routine. For instance, a *LocalSet* event updates a virtual variable, while a *NewObject* event allocates virtual heap space.

The critical responsibility of the `TraceCollector` is to determine the exact moment to stop this ingestion loop and yield a state snapshot to the frontend. To achieve a traditional stepping experience, the collector aggregates events until it encounters a strictly defined logical execution boundary. This ingestion loop terminates under the following conditions:

- The loop consumes events until it encounters a `NextLine` event indicating a transition to a *new* source code line. All intermediate mutations (e.g., evaluating subexpressions, assigning variables) that occurred on the previous line are folded into this single step.
- Execution must pause and yield the current state to the frontend whenever the program requests user input, allowing the UI to prompt the user.
- Exiting a method frame serves as a natural boundary. If the collector encounters a `MethodExit` event, it peeks at the subsequent event in the queue. If control flow is returning to the caller without immediately progressing to a new line, the collector halts to allow the user to observe the state immediately after the method's return.
- Finally, if the main method exits, the collector consumes the final event and yields a terminal `ProgramFinishedEvent`.

Once a boundary condition is met, the `TraceCollector` packages the underlying `VirtualProgramState` into an immutable data transfer object (`ImmutableVirtualProgramState`). By freezing the state, the system guarantees that the frontend receives a consistent, thread-safe snapshot of the program representing exactly one atomic step of execution flow, completely hiding the underlying complexity of the fine-grained event stream.

5.2 Constructor Event Sanitization (ConstructorReconstructor)

A significant challenge in streaming execution traces directly from source-instrumented code stems from the strict syntactic rules governing constructor chaining in object-oriented languages such as Java and C#. Specifically, any call to another constructor in the same class (i.e., `this()`) or the superclass (i.e., `super()`) must be the *first statement* in the constructor body.

Because the compiler mandates this ordering, our AST instrumenter cannot inject an `EnterConstructor` probe before the chained constructor call. (In the following, `EnterConstructor` and `ExitConstructor` denote `MethodEntry` and `MethodExit` events whose `isConstructor` flag is set.) Consequently, the execution of the chained constructor completes entirely before the parent constructor can announce its own entry. For example, consider a parameterless constructor `A()` that delegates to `A(int)` via `this(42)`. Listing 5.1 shows the original class and Listing 5.2 its (simplified) instrumented counterpart. Because `this(42)` has to remain the first statement (Listing 5.2, L4), the `EnterMethod` probe of `A()` can only be placed *after* it (L5), whereas the probes of `A(int)` (L13–L19) run in their entirety during that delegating call. Consequently, the raw event stream is emitted inside-out, as shown in the upper track of Figure 5.2.

```

1  class A {
2      int a;
3
4      A() {
5          this(42);
6      }
7
8      A(int value) {
9          a = value;
10     }
11 }

```

Listing 5.1: Original class with a constructor `A()` delegating to `A(int)` via `this(42)`.

```

1  class A {
2      int a;
3      A() {
4          this(42); // must stay first
5          Recorder.EnterMethod("A()"); // EnterConstructor
6          try {
7              Recorder.NextLine(5); // isThisOrSuper
8          } finally {
9              Recorder.ExitMethod("A()"); // ExitConstructor
10         }
11     }
12     A(int value) {
13         Recorder.EnterMethod("A(int)"); // EnterConstructor
14         Recorder.SetLocal("value", value);
15         try {
16             Recorder.NextLine(9);
17             Recorder.SetField(this, "a", a = value);
18         } finally {
19             Recorder.ExitMethod("A(int)"); // ExitConstructor
20         }
21     }
22 }

```

Listing 5.2: Simplified instrumented pseudo code of Listing 5.1 (probe arguments abbreviated). Because `this(42)` must stay the first statement, `A(int)` emits all of its events (L13–L19) before `A()` announces its own entry (L5).

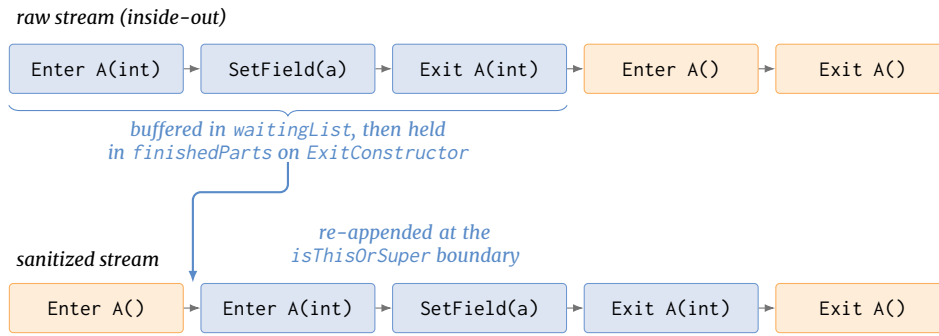


Figure 5.2: Constructor event sanitization for `A()` delegating to `A(int)` via `this(42)` (cf. Listing 5.1 and Listing 5.2). Because `this(42)` must be the first statement, the inner constructor's events (blue) are emitted before the outer constructor (orange) announces its entry. The `ConstructorReconstructor` buffers the inner sequence in `finishedParts` and re-appends it once the `NextLine` event marking the `this()` call is seen, restoring the nesting the frontend expects.

To the frontend UI and the human observer, this inverted chronological order breaks the mental model of the call stack, which conceptually enters `A()` first before pushing `A(int)` onto the stack.

To resolve this, the `TraceCollector` employs a buffered interception mechanism. Rather than forwarding events immediately to the queue, constructor-related events are diverted into the `ConstructorReconstructor`. This component acts as a specialized buffer and sequencer, utilizing two internal queues: an active `waitingList` and a `finishedParts` list.

The reconstructor sanitizes the inside-out stream through a three-phase re-ordering process, illustrated in Figure 5.2:

1. When the inner constructor (`A(int)`) executes, its entry, body, and exit events are captured in the `waitingList`. Upon receiving its `ExitConstructor` event, the reconstructor moves this entire sequence into `finishedParts`, holding the inner execution in suspension.
2. The outer constructor (`A()`) then resumes and finally emits its `EnterConstructor` event, which is placed into an empty `waitingList`. Immediately following this, the instrumented code emits a `NextLine` event indicating the boundary of the `this()` or `super()` call. The reconstructor detects this specific signature (`isThisOrSuper`). At this exact moment, it appends the suspended `finishedParts` (the inner constructor's execution) back into the `waitingList`. This effectively places the inner constructor's trace inside the outer constructor's trace, restoring the correct chronological order.
3. The reconstructor continues to buffer these reorganized events until the outer constructor exits. The `TraceCollector` detects that the constructor chain has fully resolved when it encounters a non-constructor event while holding completed parts. At this boundary, the completely sanitized, correctly nested sequence (`Enter(Outer)`, `Enter(Inner)`, ..., `Exit(Inner)`, `Exit(Outer)`) is flushed back into the main event queue for normal processing.

By resolving this compiler-induced inversion at the collection layer, the system guarantees that the downstream state simulation (`VirtualProgramState`) receives object creation events in an intuitive, hierarchical order. This ensures the frontend can accurately construct stack frames and animate complex object instantiation without visual stuttering.

5.3 The Shadow Memory Model (`VirtualProgramState`)

The raw event stream emitted by the instrumented program is purely delta-based. It broadcasts individual state changes (e.g., “local variable `x` was set to 5” or “method `foo()` was entered”). However, to render comprehensive visualizations such as a Memory Graph or a Call Stack, the frontend requires absolute state snapshots, a complete discrete picture of the program’s memory.

To bridge the gap between continuous deltas and absolute snapshots, the backend implements a state machine referred to as the `VirtualProgramState`. This component acts as a lightweight *Shadow Virtual Machine*. By sequentially applying the sanitized event chunks, it reconstructs and maintains a parallel representation of the debuggee’s memory space.

The `VirtualProgramState` is divided into three primary tracking domains:

- **The Virtual Call Stack:** The model maintains a stack of `StackFrame` objects. To accurately reflect modern language scoping rules (such as variables declared inside a while loop), each frame contains its own internal stack of variable lists. Incoming `EnterBlock` and `ExitBlock` events push and pop these sub-scopes, ensuring that shadowed variables are resolved correctly when a `LocalSet` event is processed.
- **The Virtual Heap:** The heap is simulated as a registry of dynamically allocated entities, mapped by their unique object IDs. To handle the semantic differences in memory layout, the model distinguishes between standard `VirtualObject` instances, `VirtualArray` instances (which include dynamic resizing logic to accommodate cases where the array’s initial capacity is not encoded in the `NewArray` trace event), and `VirtualString` primitives.
- **The Virtual Class Registry:** A global map tracks loaded `VirtualClass` definitions and their associated static fields, which persist independently of the call stack.

As the `TraceCollector` feeds events into the model, the `VirtualProgramState` mutates its internal structures. A `MethodEntry` event pushes a new frame; an `ArraySet` event modifies a specific index within a `VirtualArray`; and a `StaticSet` event updates the class registry.

Furthermore, the model is designed to tolerate the architectural quirks of object-oriented initialization. Because of the way `super()` constructor calls delegate execution, field assignments for an object often appear in the trace stream before the final `NewObject` event resolves the object’s concrete type. To resolve this temporal mismatch, the model employs a `tempObjects` buffer. Premature field assignments are cached in this buffer and automatically merged into the concrete `VirtualObject` once the constructor chain completes.

The last primary function of the `VirtualProgramState` is to freeze its highly mutable internal state into a thread-safe, immutable snapshot. When the overarching Step Handler determines that a logical execution boundary (such as a line step) has been reached, the model invokes its `toImmutable()` routine. This process performs a deep copy of the active stack frames, the reachable heap objects, and evaluated conditions, converting them into standardized, read-only `TraceStates`. This decoupling ensures that the frontend receives a perfectly stable, point-in-time snapshot of the execution flow, completely isolated from the continuous mutations occurring in the background trace stream.

5.4 Simulating Debugger Control Flow (Step Handlers)

In a debugger that relies on a runtime API such as *JDI*, control flow actions such as *Step Over* or *Step Out* are managed directly by the underlying runtime. Because our architecture relies on source-code instrumentation and streams data asynchronously, the debuggee executes autonomously. Consequently, standard debugger stepping mechanics cannot be enforced at the runtime level. Instead, they must be simulated within the synchronization layer by intelligently controlling the flow of events to the frontend.

To achieve this, the system models stepping operations as stream-consumption strategies. When the user requests a debug action, the corresponding `StepHandler` iteratively invokes `TraceCollector.apply()`, consuming execution events up to the next logical boundary (e.g., a line step), until a specific state condition is met. Crucially, operations that skip over code do not discard the intermediate data. Instead, they aggregate all intermediate `ImmutableVirtualProgramState` snapshots into a single `CombinedState`. This preserves the historical trace, enabling the time travel features required by the frontend UI.

The control flow operations are simulated as follows:

- **Step Into:** This is the atomic unit of advancement. The handler simply invokes `TraceCollector.apply()` exactly once, consuming the event queue until the next local execution step is reached, and yields the resulting state.
- **Step Over:** The objective of a *Step Over* action is to advance execution to the next line in the current scope, executing any subroutine calls without pausing inside them. The simulation achieves this by monitoring the call-stack depth. The handler continuously consumes trace chunks as long as the depth of the incoming state is strictly greater than the initial depth. Once the stack returns to the original depth, or lower, the handler halts and yields the combined states.
- **Step Out:** This operation advances execution until the current method frame is returned to its caller. The frontend requests this action by providing a target reference stack depth. The handler continuously consumes trace chunks until it detects a state with a stack depth less than the target depth, indicating that the current frame has been exited.

- **Run to Line and Run to End:** These operations advance execution until a specific line is reached or until the handler encounters the terminal `ProgramFinishedEvent`.

By shifting the stepping logic from runtime interrupts to state-based stream consumption, the architecture achieves two critical goals: it maintains absolute language and runtime agnosticism, and it ensures that the noise of skipped execution steps is retained in the execution history, preserving the integrity of the educational visualization.

Chapter 6

Evaluation

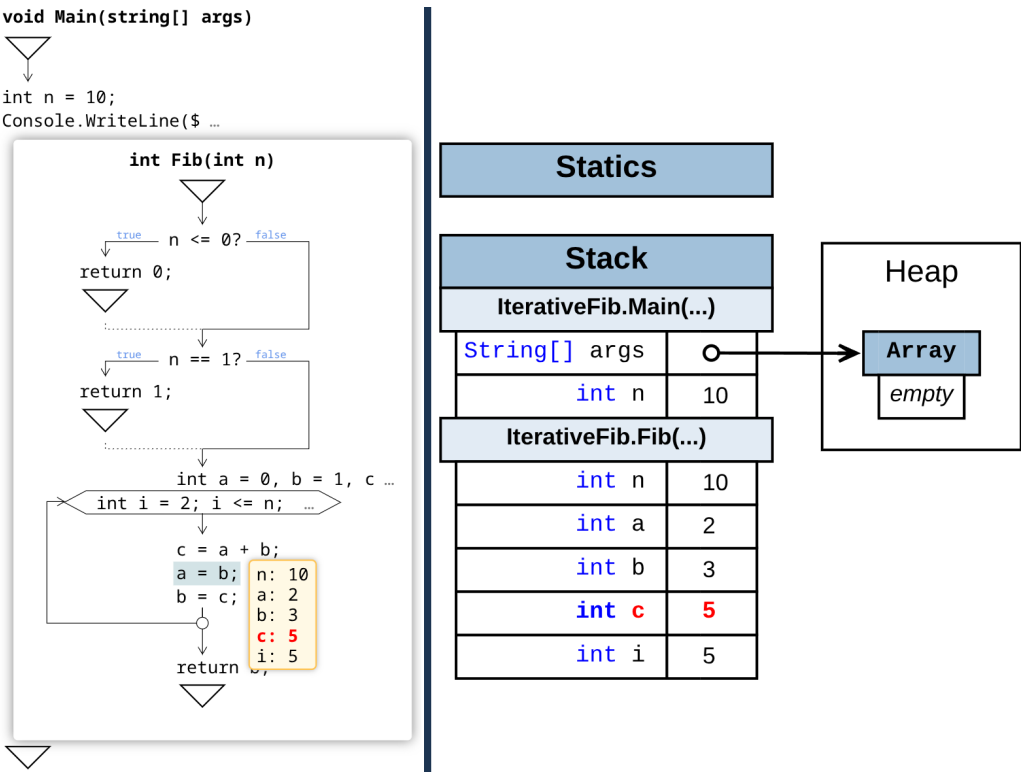


Figure 6.1: Visualization of an iterative Fibonacci implementation. We have already calculated the fifth Fibonacci number, 5 (*c*), and are preparing *a* and *b* for the sixth iteration.

To validate the architectural claims of this thesis, specifically, that a compiler-first approach enables language-agnostic debugging without platform-specific middleware, we conducted an empirical case study. We implemented a C# instrumenter using the .NET Roslyn Compiler Platform [26] and connected it to the existing, unmodified JavaWiz frontend.

The three sections of this chapter answer the research questions posed in Chapter 1 in turn: functional feasibility (RQ1, Section 6.1), the size and effort of the instrumentation code (RQ2, Section 6.2), and performance overhead (RQ3, Section 6.3). Section 6.4 then discusses the threats to the validity of these results.

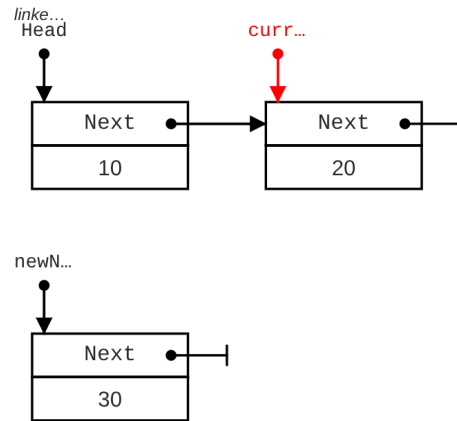


Figure 6.2: Visualization of a singly linked list.

6.1 Functional Feasibility

To answer **RQ1**, we examine whether the compiler-first approach correctly captures and reconstructs complex execution flows across languages. The primary indicator of success is behavioral correctness: the JavaWiz frontend must render visually accurate diagrams for a C# program, as demonstrated by the three programs described below. For Java, we compared the *TraceState* sequences produced by the new architecture against those produced by the legacy JDI-based backend for the same programs, which serves as a ground truth.

Control Flow and Primitives In Figure 6.1, we demonstrate the visualization of a standard iterative algorithm (calculating a Fibonacci sequence) written in C#. The frontend correctly interprets the *NextLine* and *LocalSet* events, rendering the instruction pointer and the mutation of primitive integer values. Crucially, the *Step* action in the UI advances the visualization by exactly one source-code instruction, while transparently consuming multiple events in the background.

Object References and the Heap A more rigorous test involves reference semantics. Figure 6.2 illustrates a C# singly linked list implementation. Here, the instrumenter successfully mapped C# class instantiation to *NewObject* events. The frontend renders the resulting object graph, correctly visualizing the heap pointers connecting the nodes, demonstrating that the visualization layer operates independently of the source language.

Recursion Finally, Figure 6.3 demonstrates a recursive descent parser for evaluating infix arithmetic expressions, written in C#. The visualization accurately depicts the growth and shrinkage of the call stack. Each C# method entry triggers a *MethodEntry* event, pushing a new frame to the visual stack, demonstrating that the protocol handles stack scoping independently of the underlying runtime.

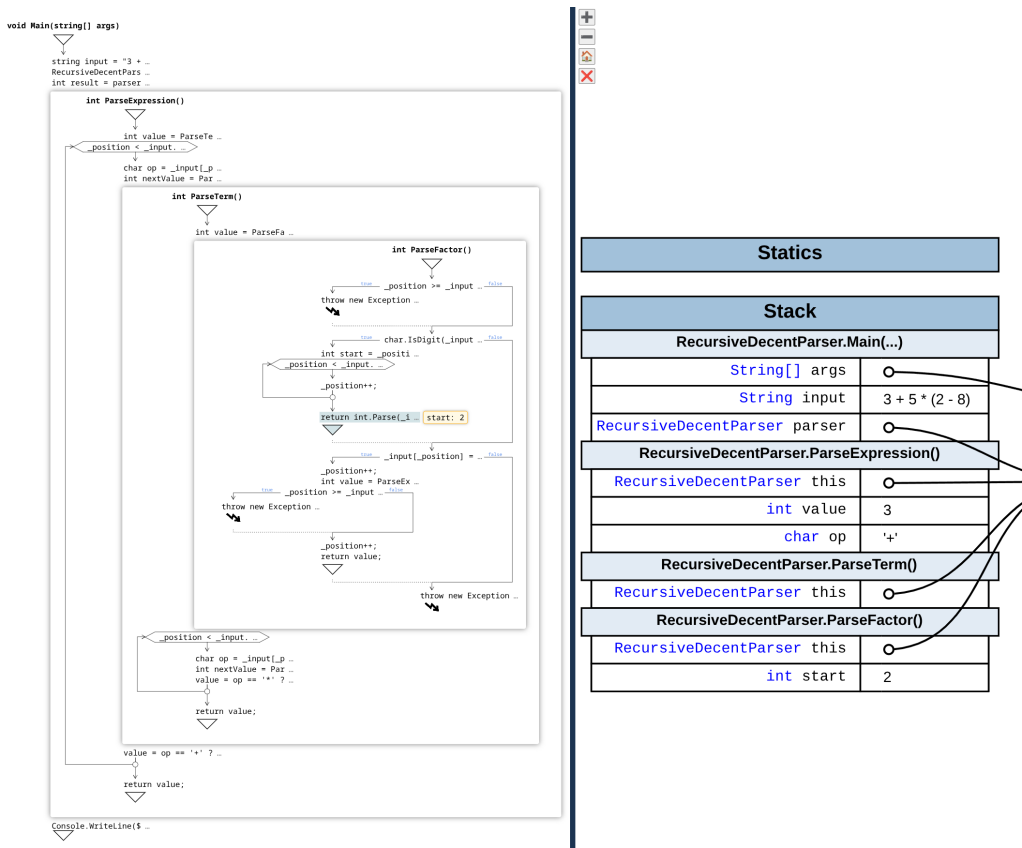


Figure 6.3: Visualization of a recursive descent parser. The parser has already consumed the first two positions (3 +), and is currently evaluating the third (5) in its fourth stack frame.

Answer to RQ1 Across iterative control flow, heap-allocated object graphs, and recursion, the unmodified JavaWiz frontend rendered correct visualizations from C# programs using the same trace protocol as Java. The compiler-first approach therefore captures and reconstructs these execution flows correctly across both languages we evaluated.

6.2 Size of the Instrumentation Code

To answer RQ2, we quantify the implementation effort required to add a new language, measured both in code size and in development time. Table 6.1 summarizes the code sizes discussed in this section.

To evaluate the code size of our Java instrumenter, we compared it to the original JDI-based JavaWiz implementation including the AST instrumentation that was already present in the original system. Note that this comparison excludes the 1,100 lines of code (LOC) of shared collection and state management infrastructure used by both the Java and C# instrumenters; this cost is amortized across all supported languages and is not attributable to any single instrumenter. The original JavaWiz implementation focusing on interaction with JDI consists of 1,500 LOC, and the AST instrumentation consists of 1,300 LOC. Our new Java instrumenter consists of 1,400 LOC, of which 900 LOC are related

Component	Part	LOC
Legacy JavaWiz (Java, JDI-based)	JDI interaction	1,500
	AST instrumentation	1,300
New Java instrumenter	AST modification	900
	Trace-emitting node creation	500
New C# instrumenter	AST modification	900
	Trace-emitting node creation	700
Shared infrastructure	Collection and state management	1,100

Table 6.1: Lines of code (LOC) of the legacy implementation, the two new instrumenters, and the shared, language-independent infrastructure.

to modifying the AST and 500 LOC are for creating new AST nodes to emit the trace events. This represents a reduction of roughly 1,400 LOC compared to the original JDI-based implementation.

The C# instrumenter, on the other hand, consists of 1,600 LOC, of which 900 LOC are related to modifying the AST and 700 LOC are for creating new AST nodes to emit the trace events.

Implementation Time The Java instrumenter took approximately three person-months to implement, primarily due to the initial implementation and continuous evolution of the trace protocol; these three months also include the time for creating the shared collection and state management code. The C# instrumenter, in contrast, took approximately 30 hours to implement, which is a significant reduction in implementation time compared to the Java instrumenter. This reduction in implementation time can be primarily attributed to the fact that for the C# instrumenter, we could reuse the protocol definition which dictates how the AST should be modified to emit the trace events.

We stress that implementation time is reported here as an observational case study rather than as a controlled quantitative metric (see Section 6.4). While this time reduction is heavily influenced by the developer’s familiarity with the protocol established during the Java implementation, it nonetheless highlights the reduced engineering overhead of the language-agnostic approach.

Answer to RQ2 Adding C# required a self-contained instrumenter of comparable size to the Java one, with the trace protocol, the state reconstructor, and the entire visualization frontend reused unchanged. The observed effort dropped from roughly three months for the first language to roughly 30 hours for the second, which indicates a substantial reduction in per-language engineering effort, subject to the caveats stated above.

6.3 Performance Considerations

To answer **RQ3**, we assess whether the overhead introduced by instrumentation and socket-based telemetry remains compatible with interactive use. Our architecture inherently introduces run-time overhead. This overhead stems from two sources:

- *Instrumentation Cost*: The injection of probe calls increases the total number of instructions executed. A simple assignment $a = b$ expands into a function call involving string allocation for the variable names and value boxing.
- *I/O Latency*: Every event requires Protobuf serialization and a TCP socket flush.

The “Human-in-the-Loop” Bound Standard debugging is an interactive process. The latency threshold for human perception is approximately 60 ms for mouse-based interaction [13]. Even with our fine-grained instrumentation, our unoptimized C# prototype processes and streams typical `LocalSet` events in under 5 μ s on higher-end consumer hardware¹.

Table 6.2 summarizes all measurements reported in this section.

Measurement	Lang.	Legacy (JDI)		New (trace-based)	
		Mean	SD	Mean	SD
Event throughput (events per ms)	Java	–	–	329	11
Session time for 300,007 <i>Step Into</i> requests	Java	106.34 s	0.676 s	36 s	0.37 s
Uninstrumented run time of the same program	Java		13.7 ms		
Instrument, compile, and start	Java	53.89 ms	2.95 ms	51 ms	14.68 ms
Instrument, compile, and start	C#	–	–	591.44 ms	5.5 ms

Table 6.2: Performance measurements for the iterative Fibonacci program: mean and standard deviation (SD) over 100 runs each, on the hardware described in Section 6.3.

To evaluate the new architecture on a basic level, we used an iterative Java Fibonacci implementation, instrumented it, and counted how many events were sent per millisecond. Through 100 consecutive tests, the instrumented application sent 329 (SD = 11) fine-grained events per millisecond on average.

More importantly, though, is how many `TraceStates`, i.e., JavaWiz’s internal data structures used on the frontend, can be reconstructed based on these fine-grained events, since multiple trace events (e.g., a method entry and all subsequent `LocalSet` events for the parameters) can be combined into a single `TraceState`. We again used the iterative Java Fibonacci implementation

¹All measurements in this section have been performed on the following system: OS: Arch Linux kernel linux-zen 6.19; CPU: AMD Ryzen 7 9800X3D; RAM: 64 GB DDR5-6000; SSD: NVMe 1 TB, 3.5 GB s⁻¹ read; Java version: JetBrains Runtime version 21; C#: .NET 9

to compute the 100,000th Fibonacci number. We attached our new debugging architecture 100 times and programmatically performed *Step Into* requests to simulate a human operator with each test run leading to the execution of 300,007 individual *Step Into* requests. On average, each run took 36 s (SD = 0.37 s). This includes the execution of the self-reporting artifact, the processing of its fine-grained events, and the reconstruction from fine-grained events to full *TraceStates* that the frontend can visualize. While this is orders of magnitude slower than a non-instrumented calculation of the same program (which took on average 13.7 ms), this is not a concern in practice, as the *Step Into* requests have to be performed by a human operator during their debugging session. Therefore, on average, the user would have to consume over 8,000 individual *Step Into* requests per second to saturate the processing capacity of our architecture. Additionally, educational programs are typically bounded in scope; for example, sorting algorithms with $N < 20$ or with a recursion depth of fewer than 10, so as not to overwhelm the students. Thus, artifacts that require several hundred *Step Into* requests are typically not used in novice teaching and therefore not our primary concern.

Run Time Comparison to the Legacy JDI-based Architecture Compared to the legacy JDI-based JavaWiz application, our new trace-based architecture required approximately one-third of the wall-clock time of the legacy system for our Fibonacci test (36 s vs. 106.34 s). Over 100 runs, the original system took on average 106.34 s (SD = 0.676 s) per run.

Compile Time Comparison to the Legacy JDI-based Architecture Regarding compile time, the mean startup times are comparable between the old and new systems for Java. To instrument, compile, and start the iterative Fibonacci program, the JDI-based JavaWiz system took on average 53.89 ms (SD = 2.95 ms). The new architecture takes 51 ms (SD = 14.68 ms) for the same Java application. The notably higher variance in the new system warrants investigation in future work. For C#, the pipeline involves additional file-system staging and an external Roslyn-based transformer invocation (producing both an intermediate AST representation and a compiled executable), taking 591.44 ms (SD = 5.5 ms) in total.

Answer to RQ3 The instrumented artifact sustains an event throughput several orders of magnitude above what a human operator can consume, and startup times remain comparable to the legacy JDI-based system for Java. In the context of pedagogical debugging, the resulting latency is therefore unlikely to negatively affect interactive use.

6.4 Threats to Validity

Our evaluation is a case study rather than a controlled experiment, and its conclusions are bounded accordingly. We discuss the four limitations we consider most consequential.

Generalizability of the Language-Agnosticism Claim We added exactly one language beyond the original Java implementation. A single additional language demonstrates that the architecture is reusable, but it does not establish that it generalizes to arbitrary languages. Java and C# are close relatives: both are statically typed, class-based, garbage-collected, and both expose a compiler front end that yields a mutable AST. Our trace protocol was designed against exactly these properties, so the effort observed for C# is plausibly a best case. Languages that violate the frontend’s ontology, for instance dynamically typed languages such as Python, languages with manual memory management and pointer arithmetic such as C++, or functional languages without a mutable stack-frame model, would exercise the semantic lowering machinery of Section 4.3 far more heavily, and we cannot extrapolate our results to them.

Implementation Time as Observational Evidence The comparison of roughly three months against roughly 30 hours is observational, not a controlled measurement. It reflects a single developer, a single attempt per language, and no comparison group. It is confounded by an unavoidable ordering effect: the Java implementation necessarily absorbed the design of the trace protocol, the shared collection infrastructure, and the state reconstructor, all of which the C# implementation inherited for free. Developer learning across the two implementations is likewise not separable from the architectural benefit we wish to attribute the reduction to. The figure should therefore be read as an indication of magnitude, not as a measured speedup.

Fibonacci as a Micro-Benchmark Our performance measurements rest on an iterative Fibonacci program. This is deliberately a micro-benchmark: it establishes the baseline event throughput of the pipeline under a dense stream of `NextLine` and `LocalSet` events, and it isolates that throughput from allocation and I/O costs. It is not representative of the full range of programs used in introductory teaching. In particular, it exercises neither heavy object instantiation, which stresses `NewObject` handling and the shadow heap in Section 5.3, nor complex or interactive I/O, which stresses the proxying described in Section 4.6. Future work should measure the overhead on programs dominated by these workloads to confirm that it remains imperceptible to the human in the loop.

Single Hardware and Software Configuration All timings were collected on one machine, with one operating system, one JVM build, and one .NET runtime version, as stated in footnote 1 of Section 6.3. We report no cross-platform or cross-runtime measurements, and we did not vary CPU, memory, or storage. The relative comparison against the legacy JDI-based system is measured on identical hardware and is therefore the more robust of our performance claims; the absolute latency and throughput figures should not be assumed to transfer to other configurations, particularly to the lower-end hardware that students may use.

Correctness Assessed by Manual Inspection The answer to RQ1 rests on multiple C# programs whose visualizations were inspected manually, together

with the comparison against the legacy backend for Java. We did not run an automated, language-independent conformance suite that checks the reconstructed `TraceStates` of both instrumenters against a shared oracle. The programs cover iteration, heap-allocated object graphs, recursion, constructor chains, and static values but not, for instance, exception handling, so correctness for those constructs is supported by the design in Chapter 5 rather than by measurement.

Chapter 7

Limitations and Future Work

While our compiler-first architecture successfully decouples visualization from platform-specific runtimes, it introduces constraints inherent to static instrumentation. In this section, we discuss these limitations and outline our roadmap for broader language support. Whereas Section 6.4 discusses threats to the validity of our *evaluation*, this chapter discusses limitations of the *approach* itself.

7.1 Limitations of Source Instrumentation

Our architectural choice to instrument source code at compile time, rather than inspecting a running process, provides significant portability. However, this decision necessarily introduces limitations in three key areas: visibility into pre-compiled code, handling of dynamic language features, and resolving semantic mismatches with the legacy frontend.

Pre-Compiled Code Unlike debugger interfaces such as *JDI/GDB* which “see” into loaded memory addresses, our approach is limited to the code we can (re-)compile. Consequently, we cannot instrument pre-compiled system libraries or third-party binaries. If a student steps into `System.Collections.Generic.List.Add()`, our visualizer treats it as a black box; even if performing a “step into”, the execution steps over the call rather than stepping through the internal logic.

However, in an educational context, this opacity is often a feature, not a bug. Pedagogical scaffolding typically aims to hide the complexity of system internals (“noise”) to focus the learners’ attention on their logic. The inability to accidentally step into pre-compiled code or code the novice has not written prevents novice confusion.

Dynamic Language Features Our instrumentation relies on modifying the *AST* at compile time. This creates blind spots for dynamic language features that are resolved only at run time, such as *Reflection* (e.g., `MethodInfo.Invoke`) or dynamic code evaluation (e.g., `eval()` in JS/Python). Since the control flow of these operations is invisible to the static analyzer, we cannot inject the necessary probes to trace their internal execution.

However, the use of reflection and metaprogramming is generally excluded from introductory programming curricula. For the target demographic of

novice learners, the lack of support for these advanced paradigms does not impact the tool's utility.

Specific C# Limitations A certain subset of C# features remains unsupported not due to instrumentation difficulties but due to the rigidity of the current visualization frontend. Because the frontend was originally modeled on the JVM, it lacks the visual primitives to represent concepts that do not exist in Java 21. These include, most notably:

- **Reference Parameters (ref/out):** The frontend uses pointers to reference heap objects exclusively. It currently cannot render the stack-to-stack references required to visualize C# ref parameters (pointers to variables in a caller's stack frame). Consequently, ref logic is currently reduced to pass-by-value in the visualization.
- **Local Functions:** C# allows methods to be defined within other methods. The frontend's data model strictly enforces a "Class→Method" hierarchy, i.e., it cannot represent a "Method→Method" scope.

7.2 Future Work

The successful application of our compiler-first architecture to C# serves as a strong validation of the model and presents promising avenues for future research aimed at broadening language support, deepening semantic fidelity, and increasing the framework's overall agnosticism.

Expansion to Interpreted Languages (Python) Our current implementation focuses on statically compiled languages (Java, C#). A key area of future work is adapting this architecture for interpreted languages such as Python. While Python lacks a compilation phase, its standard library includes the `ast` module, which allows for "just-in-time instrumentation", parsing the source, injecting probes into the AST, and compiling it to bytecode immediately before execution. We aim to demonstrate that the *Trace-ify* pipeline is equally applicable to dynamic runtimes.

Advanced C# Features The current C# instrumenter maps core imperative features (loops, classes) effectively. However, modern C# includes complex syntactic sugar such as *LINQ* (Language Integrated Query) and *Async/Await* state machines. Future iterations will focus on "de-sugaring" these constructs, visualizing a LINQ query not as a single statement but as the underlying iteration of an enumerator, and visualizing `await` calls as distinct suspension and resumption events on the call stack. Additionally, we want to focus on implementing the remaining imperative features that are currently still missing from our instrumentation and visualization, such as reference parameters and local functions.

Schema-Driven Frontend Currently, the visualization frontend retains assumptions from its Java origins (e.g., specific field layouts, the required AST abstraction). We plan to refactor the frontend to be fully *schema-driven*. In this future architecture, the initial handshake with the instrumenter will return a definition of the language’s specific memory model (e.g., “I am a C# instrumenter, I can send structs (value types) and ref params (stack-to-stack references)”). We are also planning to further generalize the AST abstraction necessary to allow for more language-specific features without having to redefine the abstraction with each additional language. This would allow the frontend to dynamically reconfigure its rendering logic, achieving more universality.

Chapter 8

Related Work

Our approach generalizes the graphical debugger JavaWiz from a Java-only tool by providing a language-agnostic architecture for educational debuggers. In this section, we present various other trace-based debuggers, i.e., debuggers able to step back in time, followed by tools for program comprehension and educational visualization tools.

Tracing Debuggers Traditional debuggers inspect the program during a live execution and can only traverse forward while accessing the current program state [2, 20, 40]. In contrast, JavaWiz is inspired by *omniscient debuggers* that allow free navigation through the execution of a program, even backwards through time. The concepts and definitions of *omniscient debuggers* [32, 33, 34, 42, 46, 47], (online and offline) *time-traveling / reverse debuggers* [11, 20, 40, 48], as well as *capture-and-replay / (record-and-)replay debuggers* [18, 29] often overlap, are mixed, and become blurred. What unites all of these approaches is that they rely on central elements of *dynamic analysis* [8]: *execution traces* [7, 9, 19, 23, 36, 45] or traces of system snapshots. Omniscient debuggers and live reverse debuggers typically provide the possibility to step back through time during a live debug session. Capture-and-replay / (record-and-)replay debuggers typically record application and system state at every program step (including external inputs) and use these to re-execute the application, with the exact same state and inputs, at a later time or even on a different machine. In contrast to these systems, which obtain their traces from the runtime or the operating system, JavaWiz obtains its trace from the program itself and is therefore not tied to a particular runtime.

Multi-Language Debugging Debugging programs that span several languages or runtimes has traditionally been solved by bridging debug interfaces. Lee et al. [21] propose portable mixed-environment debugging by composing the debuggers of the involved languages, and Ryu and Ramsey [38] achieve source-level debugging for multiple languages by keeping the language-specific part of the debugger small. Other approaches bridge managed and native code through foreign function interfaces such as JNI or P/Invoke [14, 37, 44]. More recently, the Debug Adapter Protocol [24] standardizes the interface between development tools and per-language debug adapters. All of these approaches still require a debug interface for every supported language; our architecture instead removes the debug interface entirely and makes the instrumenter the only language-specific component.

Instrumentation-Based Tracing Collecting execution traces by inserting probes into the program is a well-established technique. Instrumentation can be performed at the machine-code level [22], at the bytecode level, as done by JThreadSpy [23], or at the source level. Herber et al. [17] use traces of an instrumented bytecode interpreter to visualize program execution for compiler construction education. We deliberately instrument at the AST level (Section 4.4): lower-level instrumentation exposes compiler-generated artifacts such as bridge methods or boxing routines that have no counterpart in the learner’s mental model, whereas AST-level probes only report constructs that exist in the source code.

Visualization Tools for Program Comprehension Numerous systems and research endeavors focus on visualizing various aspects of dynamic program behavior [12, 15, 39, 41], most often also relying on various kinds of traces. For instance, Weninger et al. [49, 50, 51, 52] employ diverse visualization techniques, such as memory cities, to provide comprehension support for heap structures, aiding in understanding heap allocations and the identification of memory leaks. These tools target experts analyzing large-scale program behavior, whereas JavaWiz targets novices following the execution of small programs step by step.

Educational Visualization Tools for Programming Sorva et al. [43] give a very comprehensive tool overview for visualizations in introductory programming education. They highlight that many educational tools focus on specific program aspects or tasks such as visualizing the stack and heap memory structure visualization during program execution. For instance, JAVAVIS [30], a desktop application, offers object and sequence diagrams, pioneering memory visualization for Java. Jeliot [5, 6, 28] has a strong focus on animations [4] and provides similar functionalities in a graphically more polished fashion. The NetBeans IDE plugin EZVisor [27] visualizes program executions as abstract diagrams. jGRASP [10] is an IDE that features conceptual visualization of data with object viewers. Other examples of object structure visualization tools are Java Visualizer [16], Python Tutor [35], or VIPER [1]. All these tools inspired JavaWiz [48], a modern trace-based debugger for education that provides interactive problem-specific visualizations, including automatically generated flow charts, a detailed memory visualization, a tabular view of variable changes, animated visualizations of linked lists and binary trees, as well as 1D and 2D array operations and input buffer monitoring.

Chapter 9

Conclusion

In this thesis, we presented a framework for a language-agnostic graphical debugger that fundamentally re-imagines the relationship between the debugger and the debuggee. We demonstrated that *source code instrumentation* is not merely a fallback technique to collect additional fine-grained data, but a robust, low-barrier alternative to fully replace traditional runtime-based APIs (such as *JDI* or *ICorDebug*) for the specific domain of educational tooling.

By shifting the majority of debugging logic from the runtime to the compiler, we effectively dismantle the “language silos” that often fragment pedagogical ecosystems. We show that the language portability problem for trace-based educational visualizations can be substantially reduced within the language family considered here by a unified protocol and performing semantic lowering during the compilation phase. This approach allows divergent language features to be mapped onto a standardized visualization model without requiring complex, language-level adapters.

Our evaluation confirms that our architecture introduces less run-time overhead than the old JDI-based implementation (satisfying the performance requirements of interactive learning platforms), at the same time reducing the complexity of supporting new languages.

With respect to the research questions posed in Chapter 1, our case study yields the following answers. Regarding RQ1, the unmodified JavaWiz frontend rendered correct visualizations of iterative, heap-intensive, and recursive C# programs from the same trace protocol used for Java (Section 6.1). Regarding RQ2, adding C# required a single self-contained instrumenter of about 1,600 LOC and roughly 30 hours of observed effort, while the protocol, the state reconstructor, and the frontend were reused unchanged (Section 6.2). Regarding RQ3, the instrumented artifact sustains an event throughput several orders of magnitude above what a human operator can consume, so the resulting latency is unlikely to negatively affect interactive use (Section 6.3). Notably, the original task description anticipated modifications to the JavaWiz frontend; none were required, because the state reconstructor emits the *TraceState* format the frontend already consumes.

The limitations of the approach and the roadmap towards further languages, in particular Python, are discussed in Chapter 7.

Bibliography

- [1] Michal Adamaszek, Piotr Chrzastowski-Wachtel, and Anna Niewiarowska. 2008. VIPER, a Student-Friendly Visual Interpreter of Pascal. In *Int'l. Conf. on Informatics in Secondary Schools - Evolution and Perspectives (ISSEP) (Lecture Notes in Computer Science, volume 5090)*. Springer, pp. 192–203. DOI: 10.1007/978-3-540-69924-8_18. https://doi.org/10.1007/978-3-540-69924-8%5C_18.
- [2] Sanjeev Kumar Aggarwal and M. Sarath Kumar. 2002. Debuggers for Programming Languages. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, pp. 295–328. DOI: 10.1201/9781420040579.CH9. <https://doi.org/10.1201/9781420040579.ch9>.
- [3] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, (Eds.) ACM, pp. 572–583. DOI: 10.1145/3180155.3180175. <https://doi.org/10.1145/3180155.3180175>.
- [4] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. 2011. A decade of research and development on program animation: The Jeliot experience. *J. Vis. Lang. Comput.*, 22, 5, 375–384. DOI: 10.1016/J.JVLC.2011.04.004. <https://doi.org/10.1016/j.jvlc.2011.04.004>.
- [5] Mordechai Ben-Ari, Niko Myller, Erkki Sutinen, and Jorma Tarhio. 2001. Perspectives on Program Animation with Jeliot. In *Software Visualization, International Seminar Dagstuhl Castle (Lecture Notes in Computer Science, volume 2269)*, pp. 31–45. DOI: 10.1007/3-540-45875-1_3. https://doi.org/10.1007/3-540-45875-1%5C_3.
- [6] Sanja Maravic Cisar, Robert Pinter, and Dragica Radosav. 2011. Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3. *Int. J. Comput. Commun. Control*, 6, 4, 668–680. DOI: 10.15837/IJCCC.2011.4.2094. <https://doi.org/10.15837/ijccc.2011.4.2094>.
- [7] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. 2008. Execution Trace Analysis Through Massive Sequence and Circular Bundle Views. *J. Syst. Softw.*, 81, 12, 2252–2268. DOI: 10.1016/j.jss.2008.02.068. <https://doi.org/10.1016/j.jss.2008.02.068>.
- [8] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Trans. Software Eng.*, 35, 5, 684–702. DOI: 10.1109/TSE.2009.28. <https://doi.org/10.1109/TSE.2009.28>.

- [9] Bas Cornelissen, Andy Zaidman, Arie van Deursen, and Bart Van Rompaey. 2009. Trace Visualization for Program Comprehension: A Controlled Experiment. In *Int'l. Conf. on Program Comprehension (ICPC)*, pp. 100–109. DOI: 10.1109/ICPC.2009.5090033. <https://doi.org/10.1109/ICPC.2009.5090033>.
- [10] J.H. Cross, Dean Hendrix, and Larry Barowski. 2011. Combining Dynamic Program Viewing and Testing in Early Computing Courses. In *International Computer Software and Applications Conference (COMPSAC)*. (August 2011), pp. 184–192. DOI: 10.1109/COMPSAC.2011.31.
- [11] Pavel Dovgalyuk. 2012. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In *16th European Conf. on Software Maintenance and Reengineering (CSMR)*. Computer Society, pp. 553–556. DOI: 10.1109/CSMR.2012.74. <https://doi.org/10.1109/CSMR.2012.74>.
- [12] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. 2013. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *First Working Conf. on Software Visualization (VISOFT)*. Computer Society, pp. 1–4. DOI: 10.1109/VISOFT.2013.6650536. <https://doi.org/10.1109/VISOFT.2013.6650536>.
- [13] Valentin Forch, Thomas Franke, Nadine Rauh, and Josef F. Krems. 2017. Are 100 ms Fast Enough? Characterizing Latency Perception Thresholds in Mouse-Based Interaction. In *Engineering Psychology and Cognitive Ergonomics: Cognition and Design*. Don Harris, (Ed.) Springer International Publishing, Cham, pp. 45–56. ISBN: 978-3-319-58475-1.
- [14] Julian Garn, Florian Angerer, and Hanspeter Mössenböck. 2023. Generating Java Interfaces for Accessing Foreign Objects in GraalVM (Work in Progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*. Rodrigo Bruno and Eliot Moss, (Eds.) ACM, pp. 173–178. DOI: 10.1145/3617651.3622981. <https://doi.org/10.1145/3617651.3622981>.
- [15] Patrick Gralka, Christoph Schulz, Guido Reina, Daniel Weiskopf, and Thomas Ertl. 2017. Visual Exploration of Memory Traces and Call Stacks. In *Working Conf. on Software Visualization (VISOFT)*., pp. 54–63. DOI: 10.1109/VISOFT.2017.15. <https://doi.org/10.1109/VISOFT.2017.15>.
- [16] Philip Guo. 2013. Java Visualizer. https://cscircles.cemc.uwaterloo.ca/java_visualize. Retrieved: 2025-05-23. (2013).
- [17] Tobias Herber and Markus Weninger. [n. d.] Trace-Based Bytecode Interpreter Visualization for Compiler Construction Education. In *Working Conf. on Software Visualization (VISOFT)* – in print.
- [18] Nima Honarmand and Josep Torrellas. 2014. Replay debugging: Leveraging record and replay for program debugging. In *41st Int'l. Symposium on Computer Architecture (ISCA)*. Computer Society, pp. 445–456. DOI: 10.1109/ISCA.2014.6853229. <https://doi.org/10.1109/ISCA.2014.6853229>.
- [19] Swaminathan Jayaraman, Bharat Jayaraman, and Demian Lessa. 2017. Compact Visualization of Java Program Execution. *Softw. Pract. Exp.*, 47, 2, 163–191. DOI: 10.1002/spe.2411. <https://doi.org/10.1002/spe.2411>.

- [20] M. A. Klimushenkova and P. M. Dovgalyuk. 2017. Improving the Performance of Reverse Debugging. *Program. Comput. Softw.*, 43, 1, 60–66. DOI: 10.1134/S0361768817010042. <https://doi.org/10.1134/S0361768817010042>.
- [21] Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. 2009. Debug all your code: portable mixed-environment debugging. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA*. Shail Arora and Gary T. Leavens, (Eds.) ACM, pp. 207–226. DOI: 10.1145/1640089.1640105. <https://doi.org/10.1145/1640089.1640105>.
- [22] Jonas Maebe, Michiel Ronsse, and KD Bosschere. 2003. Instrumenting JVMs at the Machine Code Level. In *PA3CT-symposium*. Volume 19.
- [23] Giovanni Malnati, Caterina Maria Cuva, and Claudia Barberis. 2007. JThreadSpy: Teaching Multithreading Programming by Analyzing Execution Traces. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, pp. 3–13. DOI: 10.1145/1273647.1273650. <https://doi.org/10.1145/1273647.1273650>.
- [24] Microsoft. 2026. Debug Adapter Protocol - Overview. Retrieved 02/03/2026 from <https://microsoft.github.io/debug-adapter-protocol/overview%7D>.
- [25] Microsoft. 2026. LSP / LSIF. Retrieved 02/03/2026 from <https://microsoft.github.io/language-server-protocol/>.
- [26] Microsoft. 2026. The .NET Compiler Platform SDK. Retrieved 02/23/2026 from <https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/>.
- [27] Jan Moons and Carlos De Backer. 2013. The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Comput. Educ.*, 60, 1, (January 2013), 368–384. ISSN: 0360-1315. DOI: 10.1016/j.compedu.2012.08.009. <https://doi.org/10.1016/j.compedu.2012.08.009>.
- [28] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing programs with Jeliot 3. In *Working Conference on Advanced Visual Interfaces (AVI) (AVI’04)*. Gallipoli, Italy, pp. 373–376. ISBN: 1581138679. DOI: 10.1145/989863.989928. <https://doi.org/10.1145/989863.989928>.
- [29] Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *USENIX Annual Technical Conf.* USENIX Association, pp. 377–389. <https://www.usenix.org/Conf/atc17/technical-sessions/presentation/ocallahan>.
- [30] Rainer Oechsle and Thomas Schmitt. 2001. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In *Software Visualization, International Seminar Dagstuhl Castle (Lecture Notes in Computer Science, volume 2269)*, pp. 176–190. DOI: 10.1007/3-540-45875-1_14. https://doi.org/10.1007/3-540-45875-1_14.

- [31] Srđan Popić, Dražen Pezer, Bojan Mrazovac, and Nikola Teslić. 2016. Performance evaluation of using Protocol Buffers in the Internet of Things communication. In *Int'l. Conf. on Smart Systems and Technologies (SST)*, pp. 261–265.
- [32] Guillaume Pothier and Éric Tanter. 2009. Back to the Future: Omniscient Debugging. *Softw.*, 26, 6, 78–85. DOI: 10.1109/MS.2009.169. <https://doi.org/10.1109/MS.2009.169>.
- [33] Guillaume Pothier and Éric Tanter. 2008. Extending Omniscient Debugging to Support Aspect-Oriented Programming. In *Symposium on Applied Computing (SAC)*, pp. 266–270. DOI: 10.1145/1363686.1363753. <https://doi.org/10.1145/1363686.1363753>.
- [34] Guillaume Pothier, Éric Tanter, and José M. Piquer. 2007. Scalable Omniscient Debugging. In *Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 535–552. DOI: 10.1145/1297027.1297067. <https://doi.org/10.1145/1297027.1297067>.
- [35] Python Tutor. 2026. Python Tutor. Learn Java programming by visualizing code. <https://pythontutor.com/>. Retrieved: 2025-05-23. (2026).
- [36] Steven P. Reiss and Manos Renieris. 2001. Encoding Program Executions. In *International Conference on Software Engineering (ICSE)*. Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer, (Eds.) IEEE Computer Society, pp. 221–230. DOI: 10.1109/ICSE.2001.919096. <https://doi.org/10.1109/ICSE.2001.919096>.
- [37] Kenneth Russell and Lars Bak. 2001. The HotSpot™ Serviceability Agent: An Out-of-Process High-Level Debugger for a Java™ Virtual Machine. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*. USENIX Association, Monterey, CA, (April 2001). <https://www.usenix.org/conference/jvm-01/hotspot%E2%84%A2-serviceability-agent-out-process-high-level-debugger-java%E2%84%A2-virtual>.
- [38] Sukyoung Ryu and Norman Ramsey. 2005. Source-Level Debugging for Multiple Languages with Modest Programming Effort. In *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, volume 3443)*. Rastislav Bodík, (Ed.) Springer, pp. 10–26. DOI: 10.1007/978-3-540-31985-6_2. https://doi.org/10.1007/978-3-540-31985-6_2.
- [39] Anthony Savidis and Nikos Koutsopoulos. 2011. Interactive Object Graphs for Debuggers with Improved Visualization, Inspection and Configuration Features. In *International Symposium on Advances in Visual Computing (ISVC) (Lecture Notes in Computer Science, volume 6938)*, pp. 259–268. DOI: 10.1007/978-3-642-24028-7_24. https://doi.org/10.1007/978-3-642-24028-7_24.
- [40] Anthony Savidis and Vangelis Tsiatsianas. 2021. Implementation of Live Reverse Debugging in LLDB. CoRR, abs/2105.12819. arXiv: 2105.12819. <https://arxiv.org/abs/2105.12819>.

- [41] Connor Scully-Allison and Katherine E. Isaacs. 2024. Evaluating Communication Pattern Representations in Execution Trace Gantt Charts. In *Working Conf. on Software Visualization (VISSOFT)*., pp. 1–11. DOI: 10.1109/VISSOFT64034.2024.00011. <https://doi.org/10.1109/VISSOFT64034.2024.00011>.
- [42] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, and Katsuro Inoue. 2024. Evaluating the Effectiveness of Size-Limited Execution Trace with Near-omniscient Debugging. *Sci. Comput. Program.*, 236, 103117. DOI: 10.1016/J.SCICO.2024.103117. <https://doi.org/10.1016/j.scico.2024.103117>.
- [43] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.*, 13, 4, 15:1–15:64. DOI: 10.1145/2490822. <https://doi.org/10.1145/2490822>.
- [44] Dennis Strein. 2005. Design and Implementation of a high-level multi-language . NET Debugger. In *Design and Implementation of a high-level multi-language . NET Debugger*. <https://api.semanticscholar.org/CorpusID:17791697>.
- [45] Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2005. Extracting Sequence Diagram from Execution Trace of Java Program. In *Int’l. Workshop on Principles of Softw. Evolution (IWPSE)*, pp. 148–154. DOI: 10.1109/IWPSE.2005.19. <https://doi.org/10.1109/IWPSE.2005.19>.
- [46] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging. In *SIGPLAN Onward!*, pp. 89–102. DOI: 10.1145/3622758.3622892. <https://doi.org/10.1145/3622758.3622892>.
- [47] Marnix van ’t Riet, Haihan Yin, and Christoph Bockisch. 2013. The Potential of Omniscient Debugging for Aspect-Oriented Programming Languages. In *Workshop on Comprehension of Complex Systems (CoCoS)*., pp. 13–16. DOI: 10.1145/2451592.2451597. <https://doi.org/10.1145/2451592.2451597>.
- [48] Markus Weninger, Simon Grünbacher, and Herbert Prähofer. 2025. JavaWiz: A Trace-Based Graphical Debugger for Software Development Education. In *33rd IEEE/ACM Int’l. Conf. on Program Comprehension (ICPC@ICSE 2025)*. IEEE, pp. 1–12. DOI: 10.1109/ICPC66645.2025.00023. <https://doi.org/10.1109/ICPC66645.2025.00023>.
- [49] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Heap Evolution Analysis Using Tree Visualizations. In *Symposium on Software Performance (SSP)*.
- [50] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Working Conference on Software Visualization (VISSOFT)*, pp. 110–121. DOI: 10.1109/VISSOFT51673.2020.00017. <https://doi.org/10.1109/VISSOFT51673.2020.00017>.
- [51] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations. In *Smart Tools and Apps in Computer Graphics (STAG)*, pp. 63–75. DOI: 10.2312/STAG.20201241. <https://doi.org/10.2312/stag.20201241>.

- [52] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2019. Memory Leak Visualization using Evolving Software Cities. In *Symposium on Software Performance (SSP)*.

Appendix A

Supplementary Materials

This appendix provides three supplementary materials supporting the main text. Section A.1 presents the complete Protocol Buffer schema for the execution trace protocol (Chapter 4). Section A.2 provides a Java instrumentation example complementing the C# example in Chapter 4. Section A.3 presents the raw event stream produced by a single execution of that example program, making the pipeline of Chapter 5 concrete.

A.1 Execution Trace Protocol: Protobuf Schema

Listing A.1 presents the Protocol Buffer (proto3) schema that constitutes the language-agnostic contract between the language-specific instrumenters and the backend state reconstructor. Any language instrumenter that emits `TraceEvent` messages conforming to this schema integrates with the existing JavaWiz backend without modification.

```
1 syntax = "proto3";
2
3 package jwdebug.trace.data;
4
5 option java_multiple_files = true;
6 option java_package = "$$jwdebug.trace.data";
7
8 import "google/protobuf/timestamp.proto";
9
10 // Main message containing any trace event
11 message TraceEventProto {
12     EventType type = 1;
13     google.protobuf.Timestamp timestamp = 2;
14     int64 thread_id = 3;
15
16     // One of the specific event types
17     oneof event {
18         MethodEventProto method_event = 4;
19         LocalVariableEventProto local_event = 5;
20         FieldVariableEventProto field_event = 6;
21         StaticVariableEventProto static_event = 7;
22         ArraySetEventProto array_event = 8;
23         ConditionEventProto condition_event = 9;
24         ObjectCreationEventProto object_creation_event = 10;
25         ArrayCreatedEventProto array_created_event = 12;
```

```

26     StringCreationEventProto string_creation_event = 13;
27     PrintEventProto print_event = 14;
28     RegisterClassEventProto register_class_event = 17;
29     // Used for INFORM_NEXT_LINE events
30     StepEventProto step_event = 18;
31     ArrayGetEventProto array_get_event = 19;
32 }
33
34 enum EventType {
35     METHOD_ENTRY = 0;
36     METHOD_EXIT = 1;
37     FIELD_SET = 2;
38     LOCAL_SET = 3;
39     STATIC_SET = 4;
40     RETURN_VALUE = 5;
41     CONDITION = 6;
42     OBJECT_CREATION = 7;
43     ARRAY_SET = 8;
44     ARRAY_GET = 19;
45     ARRAY_CREATION = 9;
46     PRINT = 10;
47     READ = 11;
48     REGISTER_CLASS = 12;
49     STEP = 13;
50     BLOCK_ENTRY = 14;
51     BLOCK_EXIT = 15;
52 }
53 }
54
55 // For method entry/exit events
56 message MethodEventProto {
57     string class_name = 1;
58     string method_name = 2;
59     bool is_constructor = 3;
60     // only relevant for method exit events
61     int32 method_id = 4;
62     bool calledFromFinallyBlock = 5;
63     // only relevant for method entry events
64     string signature = 6;
65     string displaySignature = 7;
66     uint32 lineNumber = 8;
67 }
68
69 // For variable-related events
70 message LocalVariableEventProto {
71     string name = 1;
72     ValueProto value = 2;
73 }
74
75 message FieldVariableEventProto {
76     string name = 1;
77     ValueProto value = 2;
78     int32 owner = 3;
79 }
80

```

```
81 message StaticVariableEventProto {
82     string name = 1;
83     ValueProto value = 2;
84     // Class where the static variable is defined
85     string class_name = 3;
86 }
87
88 message ArraySetEventProto {
89     int32 array_id = 1; // ID of the array object
90     int32 index = 2; // Index in the array
91     ValueProto value = 3; // Value being set at that index
92 }
93
94 // For miscellaneous events
95 message ConditionEventProto {
96     string condition = 1; // Condition expression
97     bool value = 2; // Value of the condition
98     int32 condition_id = 3; // Unique ID for the condition
99 }
100
101 message ObjectCreationEventProto {
102     string class_name = 1; // Class of the created object
103     int32 object_id = 2; // hash code
104 }
105
106 message StringCreationEventProto {
107     string value = 1; // The string value being created
108     int32 object_id = 2; // hash code of the string object
109     // ID of the faked character array backing the string
110     int32 value_array_id = 3;
111 }
112
113 message ArrayCreatedEventProto {
114     int32 array_id = 1; // ID of the created array object
115     string class_name = 2; // Class of the array elements
116     int32 length = 3; // Length of the array
117 }
118
119 message PrintEventProto {
120     string message = 1; // The message being printed
121 }
122
123 message RegisterClassEventProto {
124     string class_name = 1;
125     int32 class_id = 2;
126     string file_name = 3;
127 }
128
129 message StepEventProto {
130     uint32 line_number = 1;
131     bool is_this_or_super = 2;
132 }
133
134 message ArrayGetEventProto {
135     int32 array_id = 1;
```

```

136     int32 index = 2;
137 }
138
139 message ValueProto {
140     string type = 1; // Type name
141
142     oneof value {
143         bool bool_value = 2;
144         int32 int_value = 3;
145         int64 long_value = 4;
146         float float_value = 5;
147         double double_value = 6;
148         string string_value = 7;
149         int32 object_id = 8; // For object references (hash code)
150         bool null_value = 9; // To represent null
151         int32 short_value = 10;
152         int32 char_value = 11;
153         int32 byte_value = 12;
154     }
155 }

```

Listing A.1: Proto3 schema for the execution trace protocol.

A.2 Java Instrumentation Example

Listing A.2 shows the Java equivalent of the C# program from Listing 4.1, and Listing A.3 shows its instrumented form. The structural similarity with the C# counterpart (Listing 4.2) reflects the unified instrumentation strategy; the syntactic differences reflect the respective compiler APIs:

- `Example.class` replaces `typeof(Example)` for class tokens.
- `Void.TYPE` replaces `typeof(void)` for the return type.
- Java's static `{ }` initializer block replaces C#'s static `Example() { }`.

```

1 class Example {
2     public static void main(String[] args) {
3         int[] array = new int[5];
4         for (int i = 1; i <= 5; i++) {
5             array[i - 1] = i * i;
6         }
7     }
8 }

```

Listing A.2: The original Java program, equivalent to Listing 4.1.

```

1 class Example {
2     public static void main(String[] args) {
3         Recorder.EnterMethod(Example.class, "main", 2,
4             "void_main(String[])", Void.TYPE, String[].class);
5         Recorder.SetLocal("args", Recorder.NewArray(args));
6         try {

```

```

7      Recorder.NextLine(3);
8      int[] array = Recorder.SetLocal("array",
9          Recorder.NewArray(new int[5]));
10     Recorder.NextLine(4);
11     Recorder.EnterBlock();
12     for (
13         int i = Recorder.SetLocal("i", 1);
14         Recorder.Condition(0, i <= 5, "i<=5");
15         Recorder.SetUnaryLocal("i", i++, i)
16     ) {
17         Recorder.NextLine(5);
18         Recorder.SetArray(array, i - 1, i * i, 0);
19     }
20     Recorder.ExitBlock();
21     Recorder.NextLine(7);
22 } finally {
23     Recorder.ExitMethod(Example.class, "main", 1, true);
24 }
25 }
26
27 static {
28     Recorder.RegisterType(Example.class, "Example.java");
29     System.setIn(new TracingInputStream(System.in));
30     System.setOut(new TracingPrintStream(System.out));
31 }
32 }

```

Listing A.3: Instrumented Java, showing injected line steps and variable tracking, based on Listing A.2.

A.3 End-to-End Trace Example

Listing A.4 presents the complete raw event stream emitted by the instrumented C# program from Listing 4.1 during a single execution. Events are listed in the order they arrive at the backend socket. Line numbers refer to the original, uninstrumented source. Iterations 3–5 of the loop body are elided; they follow the same pattern as iterations 1 and 2.

```

1 RegisterClass { class = "Example", file = "Example.cs" }
2 MethodEntry { class = "Example", method = "Main", line = 2,
3     display = "Main()", isConstructor = false }
4 NextLine { line = 4 }
5 NewArray { id = 1, type = "int", className = "Array",
6     length = 5 }
7 LocalSet { name = "array",
8     value = { type = "System.Int32[]", objectId = 1 } }
9 NextLine { line = 5 }
10 BlockEntry
11 LocalSet { name = "i", value = { type = "int", intValue = 1 } }
12 Conditional { id = 0, value = true, expr = "i <= 5" } // iter. 1
13 NextLine { line = 7 }
14 ArraySet { id = 1, index = 0,

```

```
15         value = { type = "int", intValue = 1 } }
16 LocalSet   { name = "i", value = { type = "int", intValue = 2 } }
17 Conditional { id = 0, value = true, expr = "i <= 5" } // iter. 2
18 NextLine   { line = 7 }
19 ArraySet   { id = 1, index = 1,
20             value = { type = "int", intValue = 4 } }
21 LocalSet   { name = "i", value = { type = "int", intValue = 3 } }
22 ...
23 Conditional { id = 0, value = false, expr = "i <= 5" } // end
24 BlockExit
25 NextLine   { line = 9 }
26 MethodExit { class = "Example", method = "Main" }
```

Listing A.4: Raw event stream for one execution of Listing 4.1. Iterations 3–5 are elided.