# Out of the Loop No More: Online Abstract Debugging with Loop Stepping

**João Alvim**
University of Minho, Braga, Portugal

**Maarten Steevens**
Ghent University, Belgium

**Christophe Scholliers**
Ghent University, Belgium

## Abstract

Traditional debugging makes it practically impossible to manually explore all execution paths, often leaving hypotheses insufficiently explored. Abstract debugging addresses this problem by safely over-approximating all program behaviours through a familiar stepping interface. However, existing approaches operate *offline*, pre-computing fixed loop summaries that developers cannot interactively refine or inspect at the individual iteration level.

In this paper, we propose *online abstract debugging*, which performs abstract interpretation on demand as the developer steps through the program. This novel architecture enables *loop stepping*: developers can step into a loop body for precise, iteration-specific states, or step-over to obtain a coarser summary. By interleaving analysis with execution, developers gain direct control over the cost-precision trade-off, computing high precision only where needed to settle a hypothesis. We instantiate this model for WebAssembly and present a prototype implementation, demonstrating that developers can effectively draw sound conclusions about complex, potentially non-terminating looping programs.

**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging

**Keywords and phrases** Debugging, Abstract Interpretation

**Digital Object Identifier** 10.4230/OASIcs.SLATE.2026.20

**Category** Short Paper

**Supplementary Material**
*Software (Prototype)*: `https://live-abstract-debugger.surge.sh`

**Funding** *Maarten Steevens*: Funded by the Research Foundation Flanders, grant number 1SA6C26N.

## 1 Introduction

Debugging is a highly time-consuming process [1], but traditional debuggers are limited to single concrete executions, making it laborious to thoroughly test loops and explore all possible program behaviours [20]. Abstract debugging [9] addresses this by allowing developers to step through abstract states that safely over-approximate all executions simultaneously using a familiar stepping interface. However, existing abstract debuggers operate *offline*: a static analyser pre-computes fixed summaries for the entire program before the debugging session begins. Consequently, if a loop summary loses precision, the developer cannot interactively explore individual iterations on the fly, they are forced to either accept the coarse results or restart the entire analysis (which needs to run to completion) with adjusted global parameters, such as increased loop unrolling. Overcoming these opaque workflows by focusing on usability and explainability is crucial, as poor developer experience frequently drives users to abandon static analysis tools [5].


© João Alvim, Maarten Steevens, and Christophe Scholliers;
licensed under Creative Commons License CC-BY 4.0

To overcome this, we propose *online abstract debugging*, where abstract interpretation is performed on demand as the user steps through the program. This architectural shift directly enables *loop stepping*. Developers can step into a loop to explore precise, iteration-specific abstract states, or step-over to obtain coarser loop summaries. By interleaving analysis with interaction, our approach grants the developer interactive control over the cost-precision trade-off, computing high precision only where a hypothesis demands it, without committing to a full analysis of all iterations and while preserving soundness throughout.

In this work in progress paper we detail how such an abstract debugger can be build formally and report on an initial implementation of this model for WebAssembly [8], a low-level stack-based compilation target widely used as a backend for languages [19] as diverse as Python, Go, and Haskell. We make the following contributions:

- We propose online abstract debugging and identify loop stepping as its principal consequence, extending the abstract debugging model of [9] with interactive loop unrolling.
- We present a *recipe* for systematically defining a live abstract debugger starting from a small-step operational semantics, which we illustrate using a simple WHILE language.
- We report on an initial implementation of our technique for debugging WebAssembly programs.

## 2 Online Abstract Debugging by Example

Before diving into the formal details, we first illustrate the core concepts of our online abstract debugger through a motivating example [1]. When using a traditional debugger, developers are limited to inspecting the state of a single, specific execution path, leaving them blind to how the program might behave across all possible executions. Imagine opening a standard interactive debugger, but instead of the variables pane showing exact values like $x = 4$, it displays abstractions of the concrete state, for example *intervals* representing all possible values a variable might take, such as $x \in [0, 10]$.

It allows developers to interactively step through all possible program executions at once by means of abstract interpretation [7]. To guarantee that the analysis does not get stuck in infinite loops, abstract interpretation engines use a technique called *widening*. If a variable's bounds keep growing during a loop, widening aggressively assumes the bounds might grow forever, pushing them to infinity (e.g. $+\infty$). While this guarantees the analyzer finishes, it often results in a loss of precision for the user.

To illustrate our debugger, consider the following program in the WHILE language. A developer reads an unknown integer $z$ from the user, then runs a short loop that overwrites $z$ with the constant 4 and increments a counter $x$:

$$\textbf{input}(z)\ ;\ x := 0\ ;\ \textbf{while}(x < 2)\{\ z := 4\ ;\ x := x + 1\ \}$$

Starting from an unknown value of $z$ and $x = 0$, the loop runs exactly twice ($x$ passes through 1 and then 2), assigning $z := 4$ on every iteration. Although this program is trivially simple, let us pretend that a developer is actively debugging it to verify its behavior.

### Exploring abstract states

Suppose the developer places a breakpoint at the loop head. In the variables pane they see $x \in [0, 0]$ and $z \in [-\infty, +\infty]$. They want to quickly skip past the loop, so they click the **Step Over** button.

[1] Try out this example here: `https://live-abstract-debugger.surge.sh`

Behind the scenes, the debugger hands control to the automated analyzer to summarize the entire loop. The analyzer joins the pre-loop state $x \in [0, 0]$ with the post-body state $x \in [1, 1]$ at the loop head, obtaining $x \in [0, 1]$. Fearing an infinite loop, widening kicks in, pushing the assumed upper bound to $+\infty$: the loop-head invariant becomes $x \in [0, +\infty]$. At the same loop head, $z \in [-\infty, +\infty]$ (the unknown input value) is joined with $z \in [4, 4]$ (the post-body value), yielding $z \in [-\infty, +\infty]$. When the loop terminates, the debugger applies the exit condition $x \geq 2$.

The execution marker jumps to the end of the program, and the user looks at the variables pane. It now reads $x \in [2, +\infty]$ and $z \in [-\infty, +\infty]$. The analysis is perfectly safe and mathematically sound, but the information that the loop always sets $z$ to exactly 4 has been completely lost.

#### Gaining precision by manual unrolling

Realizing that $z \in [-\infty, +\infty]$ is too imprecise to be useful, the developer restarts the debugger. This time, arriving at the loop head with $x \in [0, 0]$ and $z \in [-\infty, +\infty]$, they decide to guide the analyzer manually by clicking **Step Into**.

By manually stepping through the code, the user forces the debugger to evaluate exact bounds step-by-step, preventing the automated widening heuristic from triggering. Because $x \in [0, 0]$ is a singleton interval, the guard $x < 2$ is unambiguously true and the assignment $z := 4$ executes without any branching. The user watches the variables pane update in real-time: after the first iteration, the state becomes $x \in [1, 1]$ with $z \in [4, 4]$, and after the second iteration, it updates to $x \in [2, 2]$ with $z \in [4, 4]$.

At this point the developer steps once more to evaluate the loop guard ($x < 2$). Because the debugger knows exactly that $x \in [2, 2]$, it determines that the loop condition is unambiguously false. The debugger exits the loop naturally.

The final state in the variables pane is exactly $x \in [2, 2]$ and $z \in [4, 4]$. By simply interacting with the debugger as they would in any standard IDE, the developer acts as a human loop unroller and successfully bypasses the precision loss of widening.

This highlights the core novelty of our approach: *online abstract debugging*. Whereas previous abstract debuggers operate offline our tool performs abstract interpretation on demand over a live program. This enables *loop stepping*, granting the developer direct, interactive control over the cost-precision trade-off.

## 3 Recipe: Defining an Online Abstract Debugger

In this section, following the approach to abstract interpretation outlined in [15], we present an overview of how to construct an online abstract debugger for a small imperative programming language: WHILE. We build the formal model of the debugger in three stages: first, we establish the *collecting* small-step operational semantics of the WHILE language. Next, we lift this execution model to define an abstract interpreter and finally, we formalize our interactive debugger semantics on top of this abstract semantics.

### 3.1 The while Language

Given the intuition for our online abstract debugger, we proceed to formalize its underlying mechanics. We use the WHILE language, a small didactical imperative language, to illustrate the concept of online abstract debugging. The syntax of the WHILE language is shown in Figure 1.

$$C ::= \textbf{skip} \mid C; C \mid \mathrm{x} := E \mid \textbf{input}(\mathrm{x}) \mid \textbf{if}(B)\{C\}\textbf{else}\{C\} \mid \textbf{while}(B)\{C\}$$
$$E ::= n \mid \mathrm{x} \mid E + E$$
$$B ::= \textbf{true} \mid \textbf{false} \mid E < E \mid E = E$$
$$P ::= C$$

**Figure 1** Syntax of the WHILE language.

$$\frac{}{\langle \mathrm{x} := E, m\rangle \to \langle \textbf{skip}, m[\mathrm{x} \mapsto \mathit{eval}_E(m)]\rangle}\ (\text{E-ASSIGN})$$

$$\frac{z \text{ is an input integer}}{\langle \textbf{input}(\mathrm{x}), m\rangle \to \langle \textbf{skip}, m[\mathrm{x} \mapsto z]\rangle}\ (\text{E-INPUT}) \qquad \frac{\langle C_1, m\rangle \to \langle C_1', m'\rangle}{\langle C_1; C_2, m\rangle \to \langle C_1'; C_2, m'\rangle}\ (\text{E-SEQ-STEP})$$

$$\frac{}{\langle \textbf{skip}; C_2, m\rangle \to \langle C_2, m\rangle}\ (\text{E-SEQ-FINISH}) \qquad \frac{\mathit{eval}_B(m) = \textbf{true}}{\langle \textbf{if}(B)\{C_1\}\textbf{else}\{C_2\}, m\rangle \to \langle C_1, m\rangle}\ (\text{E-IF-TRUE})$$

$$\frac{\mathit{eval}_B(m) = \textbf{false}}{\langle \textbf{if}(B)\{C_1\}\textbf{else}\{C_2\}, m\rangle \to \langle C_2, m\rangle}\ (\text{E-IF-FALSE})$$

$$\frac{}{\langle \textbf{while}(B)\{C\}, m\rangle \to \langle \textbf{if}(B)\{C; \textbf{while}(B)\{C\}\}\textbf{else}\{\textbf{skip}\}, m\rangle}\ (\text{E-WHILE})$$

**Figure 2** Small-Step Operational Semantics of the WHILE Language.

We differentiate between commands and expressions. Commands ($C$) are state-modifying statements that dictate the control flow of the program and update the environment, whereas expressions evaluate to specific values without producing side effects. Commands ($C$) consist of standard constructs including no-operations (**skip**), sequences ($C; C$), assignments ($\mathrm{x} := E$), inputs (**input**(x)), conditionals (**if**), and loops (**while**). Arithmetic expressions ($E$) and boolean expressions ($B$) are standard. Finally, a complete program ($P$) is defined as a single top-level command $C$.

## Step1: Small-Step Operational Semantics

We define the small-step operational semantics of the WHILE language in figure 2. The evaluation relation $\langle C, m\rangle \to \langle C', m'\rangle$ transitions a configuration of a command $C$ and memory state $m$ by a single computational step, terminating at the **skip** command. For brevity, expression evaluations ($\mathit{eval}_E$ and $\mathit{eval}_B$) are abstracted as deterministic, side-effect-free functions.

The rules operate as follows: (E-ASSIGN) and (E-INPUT) update the memory $m$ with an evaluated expression or external input $z$, respectively, reducing to **skip**. Sequential composition (E-SEQ-STEP, E-SEQ-FINISH) evaluates the left command until it reduces to **skip**, then advances to the right. Conditionals (E-IF-TRUE, E-IF-FALSE) perform standard branching based on the boolean guard. Finally, (E-WHILE) defines loop execution via unrolling, expanding into an **if** command that executes the body and repeats, or terminates.

## Step2: Abstract Semantics

Given the small-step operational semantics of our language, we can proceed to define the abstract semantics. We establish this by lifting our evaluation to operate over an abstract memory $M^\sharp$ (for example, a memory of intervals) rather than a concrete memory $m$. The abstract transition relation $\langle C, M^\sharp\rangle \to^\sharp \langle C', M'^\sharp\rangle$ maps a configuration to a valid successor configuration.

The rules adapt their concrete counterparts to the abstract domain: (A-ASSIGN) evaluates expressions using an abstract evaluation function $eval^\sharp_E$ and modifies the state via $update^\sharp$. Because user input is entirely unknown, (A-INPUT) safely over-approximates it by assigning the top element $\top$. Sequential composition (A-SEQ-STEP, A-SEQ-FINISH) propagates the transition directly.

For branching, note that both (A-IF-TRUE) and (A-IF-FALSE) might apply simultaneously if the abstract state cannot definitively prove the condition true or false. For example, the condition $x > 0$ given $x \in [-5, 5]$ can evaluate to both true and false. To handle this, these rules rely on a $filter^\sharp$ function that effectively splits the abstract memory into two disjoint parts: one encompassing the states that satisfy the condition, and another for those that do not. A transition along a given branch is only valid if its corresponding filtered memory remains feasible (i.e., $\neq \bot$).

Finally, (A-WHILE) handles loops via unrolling.

**Abstract Evaluation** $\langle C, M^\sharp\rangle \to^\sharp \langle C', M'^\sharp\rangle$

$$\frac{}{\langle \mathrm{x} := E, M^\sharp\rangle \to^\sharp \langle \mathbf{skip}, update^\sharp_{\mathrm{x}}(M^\sharp, eval^\sharp_E(M^\sharp))\rangle}\ (\text{A-ASSIGN})$$

$$\frac{}{\langle \mathbf{input}(\mathrm{x}), M^\sharp\rangle \to^\sharp \langle \mathbf{skip}, update^\sharp_{\mathrm{x}}(M^\sharp, \top)\rangle}\ (\text{A-INPUT})$$

$$\frac{\langle C_1, M^\sharp\rangle \to^\sharp \langle C_1', M'^\sharp\rangle}{\langle C_1; C_2, M^\sharp\rangle \to^\sharp \langle C_1'; C_2, M'^\sharp\rangle}\ (\text{A-SEQ-STEP}) \qquad \frac{}{\langle \mathbf{skip}; C_2, M^\sharp\rangle \to^\sharp \langle C_2, M^\sharp\rangle}\ (\text{A-SEQ-FINISH})$$

$$\frac{filter^\sharp_B(M^\sharp) \neq \bot}{\langle \mathbf{if}(B)\{C_1\}\mathbf{else}\{C_2\}, M^\sharp\rangle \to^\sharp \langle C_1, filter^\sharp_B(M^\sharp)\rangle}\ (\text{A-IF-TRUE})$$

$$\frac{filter^\sharp_{\neg B}(M^\sharp) \neq \bot}{\langle \mathbf{if}(B)\{C_1\}\mathbf{else}\{C_2\}, M^\sharp\rangle \to^\sharp \langle C_2, filter^\sharp_{\neg B}(M^\sharp)\rangle}\ (\text{A-IF-FALSE})$$

$$\frac{}{\langle \mathbf{while}(B)\{C\}, M^\sharp\rangle \to^\sharp \langle \mathbf{if}(B)\{C; \mathbf{while}(B)\{C\}\}\mathbf{else}\{\mathbf{skip}\}, M^\sharp\rangle}\ (\text{A-WHILE})$$

**Figure 3** Abstract operational semantics of the WHILE language.

**Abstract Step Function and Analysis.** While the abstract semantics describes how a single configuration evolves, a static analysis must track the evolution of the entire program state. We define an *abstract step function* $Step^\sharp(\mathcal{S}^\sharp)$ that propagates a set of abstract configurations $\mathcal{S}^\sharp$. This function collects all possible successors ($\langle C, M^\sharp\rangle \to^\sharp$), for a single command $C$, as visualized in Figure 4 (Right) and joins the abstract memories of configurations, i.e. $M'_T$ and $M'_F$.

**Algorithm:** FIX

1: $\mathcal{S}^\sharp \leftarrow \{\langle P, \bot \rangle\}$
2: **repeat**
3: $\quad \mathcal{R}^\sharp \leftarrow \mathcal{S}^\sharp$
4: $\quad \mathcal{S}^\sharp \leftarrow \mathcal{R}^\sharp \cup^\triangledown \mathit{Step}^\sharp(\mathcal{R}^\sharp)$
5: **until** $\mathcal{S}^\sharp \sqsubseteq \mathcal{R}^\sharp$
6: **return** $\mathcal{S}^\sharp$

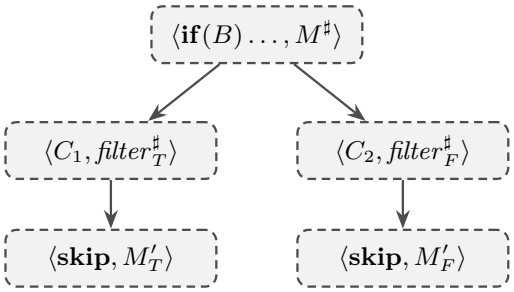


**Figure 4 (Left)** The automated fixpoint analysis algorithm using widening ($\cup^\triangledown$). **(Right)** Visualizing abstract branching where multiple feasible paths create separate frontiers.

To perform a complete analysis, we compute the least fixpoint of this step function as shown in Figure 4 (Left). However, because our interval domain has infinite height, a standard iteration might never stabilize. We therefore use a *widening operator* ($\triangledown$) to force convergence by aggressively pushing growing bounds to infinity. For an interval $[lo, hi]$, widening is defined as:

$$[lo_1, hi_1] \triangledown [lo_2, hi_2] = [(lo_2 < lo_1 ? -\infty : lo_1), (hi_2 > hi_1 ? +\infty : hi_1)]$$

This ensures that any ascending chain of abstract memories stabilizes in a finite number of steps. The complete analysis algorithm, which computes a stable set of abstract configurations $\mathcal{S}^\sharp$, is shown in Figure 4.

The widened union $\mathcal{A} \cup^\triangledown \mathcal{B}$ merges successors into the current state. If a bound at a specific program point grows, the operator applies $\triangledown$ to jump to a stable limit. This algorithm is what powers the StepOver command in our debugger: it automatically computes the entire loop summary, potentially sacrificing precision for speed.

## Step3: Online Abstract Debugger Semantics

The final step is to define the abstract debugger semantics. Rather than passively analyzing the code, these semantics must accommodate interactive commands and manage the multiple states resulting from abstract branching. The key novelty of this semantics is that it allows the user to navigate divergent branches and dynamically tune analysis precision.

**Debugger Configuration.** A debugger configuration is a tuple $\langle op, C, M^\sharp, \sigma \rangle$, where $op \in \{\textbf{Step}, \textbf{StepOver}, \textbf{StepOut}\}$ is the user's current debug operation, $C \in \mathit{Cmd}$ and $M^\sharp \in \mathbb{M}^\sharp$ form the current abstract configuration, and $\sigma$ is an execution stack used to manage branching.

When an abstract condition cannot be definitively proven true or false, the execution must split. The stack $\sigma$ enables the debugger to explore one branch while suspending alternative paths. Each stack frame $(\mathit{pending}, \mathit{acc})$ tracks a list of unexplored configurations ($\mathit{pending}$) alongside an accumulated memory ($\mathit{acc}$) that joins the abstract states of all branches that have completed execution at the current level.

**Debugger Transition Rules.** The debugger transition relation $\rightarrow_{\mathsf{dbg}}$ evaluates a given operation applied to the current state, mapping it to the resulting execution state. It is defined by the five rules in Figure 5.

$$\textbf{Debugger Transition} \qquad \langle op, C, M^\sharp, \sigma\rangle \to_{\mathsf{dbg}} \langle C', M'^\sharp, \sigma'\rangle$$

$$\frac{\text{unique } \langle C, M^\sharp\rangle \to^\sharp \langle C', M'^\sharp\rangle}{\langle \textbf{Step}, C, M^\sharp, \sigma\rangle \to_{\mathsf{dbg}} \langle C', M'^\sharp, \sigma\rangle}\ (\textsc{d-step}) \qquad \frac{}{\langle \textbf{Step}, \textbf{skip}, M^\sharp, \sigma\rangle \to_{\mathsf{dbg}} \mathit{resume}(M^\sharp, \sigma)}\ (\textsc{d-finish})$$

$$\frac{\langle C, M^\sharp\rangle \to^\sharp \langle C_1, M_1^\sharp\rangle \qquad \langle C, M^\sharp\rangle \to^\sharp \langle C_2, M_2^\sharp\rangle \qquad C_1 \neq C_2}{\langle \textbf{Step}, C, M^\sharp, \sigma\rangle \to_{\mathsf{dbg}} \langle C_1, M_1^\sharp, ([\langle C_2, M_2^\sharp\rangle], \bot) :: \sigma\rangle}\ (\textsc{d-branch})$$

$$\frac{}{\langle \textbf{StepOver}, \textbf{while}(B)\{C\}, M^\sharp, \sigma\rangle \to_{\mathsf{dbg}} \mathit{resume}\big(\mathsf{fix}(\textbf{while}(B)\{C\}, M^\sharp), \sigma\big)}\ (\textsc{d-step-over})$$

$$\frac{}{\langle \textbf{StepOut}, C, M^\sharp, (\mathit{pending}, \mathit{acc}) :: \sigma\rangle \to_{\mathsf{dbg}} \mathit{resume}(\mathsf{exhaust}(C, M^\sharp, \mathit{pending}, \mathit{acc}), \sigma)}\ (\textsc{d-step-out})$$

**Figure 5** The live debugger operational semantics.

The (D-STEP) rule handles standard, deterministic execution where the abstract memory yields a single valid successor. Conversely, (D-BRANCH) applies when the abstract memory is consistent with both outcomes of a branch condition, yielding two valid transitions. The debugger immediately steps into one branch ($C_1$) while pushing the other ($C_2$) onto the stack $\sigma$. The (D-STEP-OVER) rule applies exclusively when the current command is a **while**-loop and the user chooses to skip it. It invokes the fix helper (as described in Section 2) to instantly compute the summary, entirely bypassing manual iteration. The (D-STEP-OUT) rule runs $C$ and all pending siblings to completion with *exhaust* and calls *resume* on the parent stack, popping exactly one frame. Finally, (D-FINISH) intercepts any execution that reaches **skip** and triggers the *resume* routine to unwind the stack. The *resume* helper determines the next state by inspecting the current stack $\sigma$:

$$\mathit{resume}(M^\sharp, \sigma) = \begin{cases} \langle \textbf{skip}, M^\sharp, []\rangle & \text{if } \sigma = [] \\ \langle C', M'^\sharp, (\mathit{rest}, \mathit{acc} \sqcup M^\sharp) :: \sigma'\rangle & \text{if } \sigma = (\langle C', M'^\sharp\rangle :: \mathit{rest}, \mathit{acc}) :: \sigma' \\ \mathit{resume}(\mathit{acc} \sqcup M^\sharp, \sigma') & \text{if } \sigma = ([], \mathit{acc}) :: \sigma' \end{cases}$$

The first case signifies total termination. The second case pops the next pending branch to explore it, joining the just-finished memory $M^\sharp$ into the frame's accumulator. The third case handles an exhausted frame by joining its accumulator upward into the parent frame and recursing.

## 4 An Online Abstract Debugger for WebAssembly

We implemented the formalisation presented in the previous sections as both a concrete and abstract debugger for Miniwasm [17]. Built in TypeScript atop the `wasm-ts-runtime` engine[2], our tool supports parsing, executing, and abstract debugging both WebAssembly text files and raw binaries. The web-based interface (Figure 6) consists of three primary sections: a text editor (left) highlighting the active instruction, a state viewer (middle) dynamically displaying the abstract operand stack and local variables (e.g., intervals like $[4, +\infty]$), and an interactive console (right). A top configuration bar enables users to toggle

[2] `https://github.com/ryohey/ts-wasm-runtime`

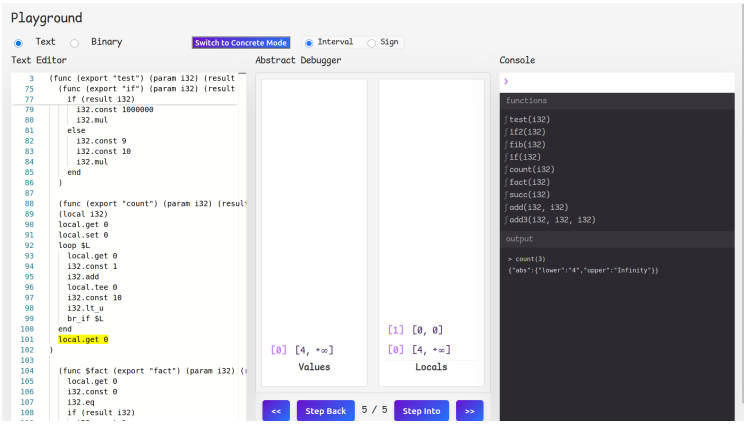


**Figure 6** The Miniwasm web interface operating in Abstract Debugger mode.

between concrete and abstract modes, select abstract domains (such as Interval or Sign), and switch input formats. Execution flow is managed via a bottom control bar, providing standard operations to step backward, step into, or advance through the program's abstract frontiers.

## 5 Related Work

The term *abstract debugging* [3] was first coined by Bourdoncle to describe an offline static analysis driven by user-defined predicates manually annotated in the source code. This is quite different from our approach, where we define an *online* abstract debugger instead of a static analysis framework. Similarly, program slicing focuses on extracting only the statements relevant to a slicing criterion [11], whereas abstract debugging focuses on inspecting sound over-approximated program states during execution.

Building on the idea of abstract debugging, Holter et al. [9] use an inter-procedural Abstract Reachability Graph (iARG) to map static threads analysis results to standard debugger interfaces. While this enables the exploration of feasible program paths, the underlying static analyzer must still compute all fixpoints and apply loop widening before the interactive session begins. Similarly, other approaches provide comprehensive offline reports [4] or interactive, web-based interfaces like Try-Mopsa [13]. Additionally, some WebAssembly tools, such as Wassail [18], WasmGrind [12], and Wastrumentation [14], have been developed, but their core analysis remains strictly offline.

Alternatively, random property-based testing [6], state space exploration techniques such as model checking used in [2], symbolic execution [10], and multiverse debugging [16] allow for interactive navigation but struggle with state and path explosion. Abstract debugging inherently mitigates this explosion by operating over abstract domains. Building on this advantage, we propose *online abstract debugging*, which replaces offline precomputation with abstract states dynamically computed on demand. This allows developers to interactively explore intermediate states, such as those inside loops, before widening occurs.

## 6 Conclusion

In this paper, we introduced *online abstract debugging*. By interleaving abstract interpretation directly with the debugging session, our architecture enables *loop stepping*. This grants developers interactive control over the cost-precision trade-off, allowing them to manually

unroll loops for iteration-specific precision or step over them using automated summaries. We formalized this approach systematically, lifting a standard small-step operational semantics to a live debugger, and demonstrated its practical viability through a web-based prototype for WebAssembly (Miniwasm). Future work will expand our model to support full WebAssembly features, integrate relational abstract domains, and evaluate the approach through an empirical user study.

## References


1 Abdulaziz Alaboudi and Thomas D. LaToza. An exploratory study of debugging episodes, 2021. arXiv:2105.02162.

2 Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL – A Tool Suite for Automatic Verification of Real–Time Systems. *BRICS Report Series*, 3(58), 1996. doi:10.7146/brics.v3i58.18769.

3 François Bourdoncle. Abstract debugging of higher-order imperative languages. *SIGPLAN Not.*, 28(6):46–55, 1993. doi:10.1145/173262.155095.

4 David Bühler, André Maroneze, and Valentin Perrelle. Abstract interpretation with the EvaEva (Frama-C plug-in) plug-in. In Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles, editors, *Guide to Software Verification with Frama-C: Core Components, Usages, and Applications*, pages 131–186. Springer International Publishing, Cham, 2024. doi:10.1007/978-3-031-55608-1_3.

5 Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, pages 332–343, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2970276.2970347.

6 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, 2000. doi:10.1145/357766.351266.

7 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. Association for Computing Machinery. doi:10.1145/512950.512973.

8 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062363.

9 Karoliine Holter, Juhan Oskar Hennoste, Patrick Lam, Simmo Saan, and Vesal Vojdani. Abstract debuggers: Exploring program behaviors using static analysis results. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '24, pages 130–146, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3689492.3690053.

10 Hao Jin, Yanyan Jiang, Na Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. Concolic metamorphic debugging. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 232–241, 2015. doi:10.1109/COMPSAC.2015.79.

11 Isabella Mastroeni and Damiano Zanardini. Abstract program slicing: An abstract interpretation-based approach to program slicing. *ACM Trans. Comput. Logic*, 18(1), 2017. doi:10.1145/3029052.

12 Peter Thiemann Michael Staab. Wasmgrind: Towards dynamic concurrency analysis for multithreaded webassembly. In *Workshop proceedings (metadata pending)*, 2025.

13 Raphaël Monat. Try-mopsa: Relational static analysis in your pocket. In Yu-Fang Chen, Thomas Jensen, and Ondřej Lengál, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 197–211, Cham, 2026. Springer Nature Switzerland. doi:10.1007/978-3-032-15700-3_10.

**14** Aäron Munsters, Angel Luis Scull Pupo, and Elisa Gonzalez Boix. Wastrumentation: Portable WebAssembly Dynamic Analysis with Support for Intercession. In Jonathan Aldrich and Alexandra Silva, editors, *39th European Conference on Object-Oriented Programming (ECOOP 2025)*, volume 333 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:29, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2025.23.

**15** Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020.

**16** Maarten Steevens, Tom Lauwaerts, and Christophe Scholliers. Concolic multiverse debugging. In *Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques*, DEBT 2024, pages 28–29, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3678720.3685318.

**17** Quentin Stiévenart and Coen De Roover. Compositional information flow analysis for webassembly programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 13–24, 2020. doi:10.1109/SCAM51674.2020.00007.

**18** Quentin Stiévenart and Coen De Roover. Wassail: a webassembly static analysis library, 2021.

**19** WebAssembly Community Group. Developer's guide – WebAssembly. https://webassembly.org/getting-started/developers-guide/. Acedido a: 18 abr. 2026.

**20** Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier/Morgan Kaufmann, Amsterdam; Boston, 2nd edition, 2009.