SolQDebug: Debug Solidity Quickly for Interactive Immediacy in Smart Contract Development

Inseong Jeon[1], Sundeuk Kim[1], Hyunwoo Kim[1], Hoh Peter In[1*]

[1*]Department of Computer Science, Korea University, 145, Anam-ro, Seonbuk-gu, 02841, Seoul, Republic of Korea.

*Corresponding author(s). E-mail(s): hoh_in@korea.ac.kr;
Contributing authors: iwwyou@korea.ac.kr; sd_kim@korea.ac.kr; khw0809@korea.ac.kr;

Abstract

Debugging Solidity contracts remains cumbersome and slow. Even a simple inspection, such as tracking a variable through a branch, requires full compilation, contract deployment, preparatory transactions, and step-by-step bytecode tracing. Existing tools operate only after execution and offer no support while code is under construction. We present SolQDebug, the first interactive, source-level assistant for Solidity developers that provides millisecond feedback before compilation or chain interaction. SolQDebug extends the Solidity grammar with interactive parsing, incrementally maintains a dynamic control-flow graph, and performs interval-based abstract interpretation guided by inline test annotations, enabling developers to simulate symbolic inputs and inspect contract behavior as in traditional debugging environments. In an evaluation on real-world functions, SolQDebug enables low-latency, statement-level analysis during development without requiring compilation or deployment.

Keywords: Smart Contract Development, Solidity, Debugging, Abstract Interpretation

1 Introduction

Smart contracts are the backbone of decentralized applications, and Solidity has become the dominant language for writing them (26**?**). As contracts grow more complex and control more assets, developers must reason about correctness throughout the development cycle—not just at deployment. Large language models (LLMs)

such as ?) or Llama (14) can assist with code generation but offer no guarantees of correctness. Ultimately, developers remain responsible for understanding variable interactions, control flow, and numeric boundaries during authoring.

Unfortunately, the debugging workflow for Solidity lags far behind traditional programming environments. Even a single inspection requires full compilation, deployment, transaction-based state setup, and manual bytecode-level tracing. Tools like Remix IDE (20), Hardhat (9), and Foundry Forge (5) replicate this costly pipeline, providing no live feedback during edits. A prior study found that 88.8% of Solidity developers described debugging as painful, and 69% attributed this to the absence of interactive, source-level tooling Zou et al. (39). Despite this widely acknowledged pain point, we find no existing research or tooling that provides interactive feedback during Solidity code authoring—a gap that this paper aims to fill.

This paper presents SolQDebug, a source-level interactive Solidity debugger powered by abstract interpretation. Rather than replacing runtime debuggers, it complements them by enabling symbolic, per-statement inspection during code authoring—before compilation or deployment. It targets the Solidity pattern of single-contract, single-transaction execution, where each function is isolated and stateless—ideal for static reasoning but difficult to simulate manually. To support this, SolQDebug applies interval-based abstract interpretation, which generalizes over symbolic inputs, exposes edge-case behaviors, and provides sound results with low overhead. This approach gives developers immediate feedback and enables them to reason efficiently about how symbolic inputs influence variable behavior. Although these inputs enable generalization across multiple cases, certain input configurations or control structures may lead to wider output ranges. We evaluate these behaviors empirically and propose annotation strategies that help maintain interpretability across typical Solidity patterns.

To achieve this goal, SolQDebug builds on two core ideas. First, it extends the Solidity grammar with interactive parsing rules and dynamically updates the control-flow graph to reflect incremental edits, enabling keystroke-level structural changes during code authoring. Second, it performs abstract interpretation seeded by inline annotations. These annotations, written directly in the source code, allow developers to specify symbolic values for both parameters and storage variables, similar to how traditional debuggers let users configure initial states and explore control flow.

We evaluate SolQDebug on real-world functions from Zheng et al. (37), demonstrating millisecond-scale responsiveness under symbolic input. Beyond latency, we analyze how input interval structure affects interpretability in common Solidity patterns, such as division-normalized arithmetic.

This paper makes the following contributions:

- We identify the main barriers to interactive Solidity debugging: latency from compilation, deployment, and transaction setup, and EVM constraints that prevent lightweight re-execution.
- We design an interactive parser and dynamic control-flow graph (CFG) engine that supports live structural updates and syntactic recovery.
- We introduce an abstract interpreter that incorporates developer annotations as symbolic input, supporting fast, deployment-free debugging workflows.

- We implement and evaluate SolQDebug on real-world contracts, demonstrating its millisecond responsiveness and exploring annotation strategies that maintain interpretability under a range of symbolic input patterns.

2 Background

2.1 Structure of Solidity Smart Contract

Solidity smart contracts may declare contracts, interfaces, and libraries. Executable business logic typically resides in contracts, and functions serve as transaction entry points. Variables are usefully grouped as global (EVM metadata such as msg.sender or block. timestamp), state (persistent storage owned by a contract), and local (scoped to a call). Types include fixed-width integers, address, booleans, byte arrays, and user-defined structs; containers include arrays and mappings. A mapping behaves like an associative array with an implicit zero value for unseen keys and is not directly iterable. Storage classes (storage, memory, calldata) indicate lifetime and mutability; we mention them only to fix terminology. Visibility and mutability qualifiers (public, external, internal, private; pure, view, payable) exist but are not central to our single-contract, single-transaction setting. Control flow (if/else, while/for/do-while, break/continue, return) follows C/Java conventions.

Listing 1: Minimal example used to illustrate grammar elements relevant to our analysis

```
contract Example {
    address public owner;
    uint256 public totalSupply = 1000;
    mapping(address => uint256) private balances;

    modifier onlyOwner() {
        require(msg.sender == owner, "not owner");
        _;
    }

    function burn(uint256 amount) public onlyOwner {
        uint256 bal = balances[msg.sender];
        uint256 delta;
        if (bal >= amount) {
            balances[msg.sender] = bal - amount;
            delta = amount;
        }
        else {
            delta = 0;
        }
        totalSupply -= delta;
    }
}
```

The example highlights the specific features we rely on later. State variables include general types (owner, totalSupply) and a mapping from addresses to balances; global

variables appear implicitly in guards via msg.sender. The function burn introduces parameters and a local variable (bal). The modifier onlyOwner performs a precondition check before the function body executes; the placeholder underscore marks where the original body is inserted when the modifier is inlined. In analysis, such modifiers are expanded at their precise positions around the function body in the control-flow graph.

These grammar elements connect directly to our semantics. Guards such as require narrow feasible ranges along taken branches. Modifiers are inlined so that their precondition checks are analyzed in sequence with the function body. Containers like mappings remain symbolic until a concrete key is accessed, at which point an abstract value is materialized for that access. This level of detail suffices for our abstract interpretation in the single-contract, single-transaction scope without introducing parts of the language that our evaluation does not exercise.

2.2 Solidity Execution Model

To execute a Solidity contract on the blockchain, it must first be deployed. Deployment occurs through a one-time transaction that stores the compiled bytecode on-chain and invokes the constructor exactly once. After deployment, all subsequent interactions are message-call transactions. In these, the caller specifies a public function along with encoded calldata. Once the transaction is mined into a block, the Ethereum Virtual Machine (EVM) jumps to the designated entry point and executes the corresponding function sequentially. At runtime, Solidity variables fall into three distinct storage classes (26):

- **Global variables** represent implicit, read-only metadata provided by the EVM, such as `block.timestamp`, `msg.sender`, and `msg.value`.
- **State variables** store persistent data within the contract and retain their values across transactions.
- **Local variables** include function parameters and temporary values scoped to a single execution context.

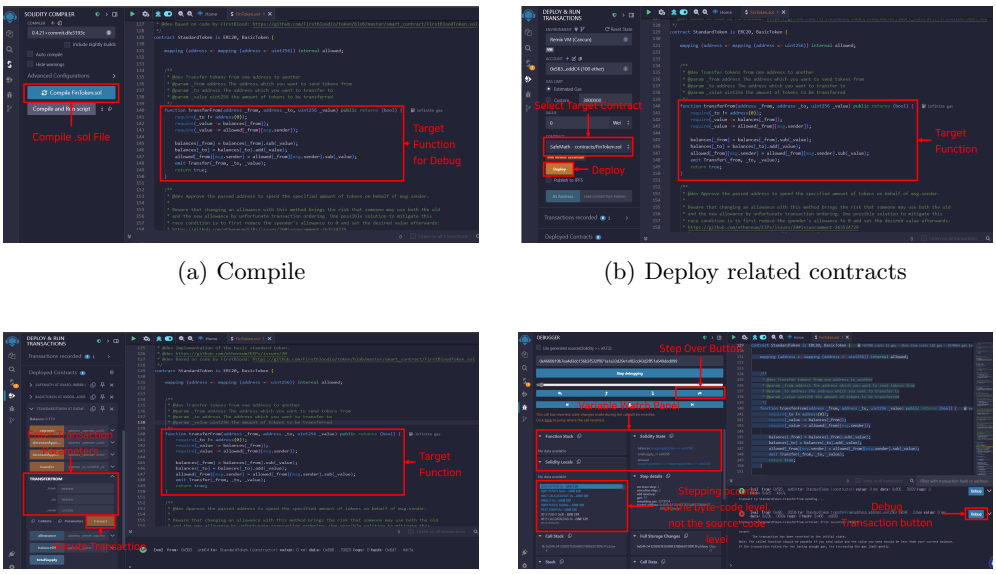
These three classes share a unified type system comprising primitive types like `uint` , `int`, `bool`, and `address`, as well as composite types such as arrays, mappings, and structs. Composite values can be nested to arbitrary depth using field access (`.`) or indexing (`[]`). Control flow follows familiar C-style constructs such as `if/else`, `while`, `for`, and `return`, alongside Solidity-specific statements like `emit` and `revert`.

As a result, debuggers must resolve potentially complex, multi-step expressions to analyze deeply nested elements within the contract state.

2.3 Root Causes of the Solidity-Debugging Bottleneck

Debugging Solidity programs remains significantly slower than traditional application development workflows due to two orthogonal obstacles.

(1) Environmental disconnect. Unlike conventional IDEs such as PyCharm (11) or Visual Studio (17), where the source editor and execution engine run in the same process, Solidity development involves external coordination with a blockchain node at every stage of the workflow. Even a single debugging cycle must pass through four



(a) Compile

(b) Deploy related contracts

(c) Send preparatory transactions

(d) Bytecode-level debugging

Fig. 1: Traditional Solidity debugging workflow

sequential stages (see Fig.1). First, the contract must be compiled. Then, the bytecode is deployed to a local or test chain. Next, developers must manually initialize the on-chain state by sending setup transactions. Finally, the target function is invoked, and its execution is traced step by step at the bytecode level.

This workflow introduces several seconds to minutes of latency per iteration, fundamentally breaking the fast "type-and-inspect" feedback cycle expected in modern development tools. To mitigate this friction, developers often rely on `emit` logs or event outputs to observe intermediate values. However, such instrumentation provides only runtime snapshots and lacks the structural insight needed to understand symbolic variation or control-flow behavior. Moreover, modifying the expression of interest typically requires recompilation and redeployment, compounding latency and disrupting iteration. The final stage—tracing raw EVM opcodes—is particularly costly, as developers are forced to mentally reconstruct source-level semantics. This not only adds execution overhead but also imposes significant cognitive burden during fault localization and fix validation.

(2) Architectural limitations of the EVM. The Ethereum Virtual Machine (EVM) is a state-based execution engine in which each transaction mutates a globally persistent storage. Once a function executes, its side effects are irreversible unless external intervention is performed. Re-executing the same function along the same control path is nontrivial: developers must either redeploy the entire contract to restore the initial state, or manually reconstruct the required preconditions via preparatory transactions—both of which incur significant overhead.

Additionally, if a function includes conditional guards that depend on the current state—such as account balances or counters—then any debugging session must first ensure that those conditions are satisfied. Fig. 2 illustrates this challenge: the debug target function enforces a check on `_balances[account]`, requiring developers to manually assign a sufficient balance before they can observe the downstream effects on `_totalSupply`. Without such setup, the function exits early, preventing inspection of the intended execution path.

In short, these constraints make repeated debugging iterations costly and fragile. According to a developer study (39), 88.8% of Solidity practitioners reported frustration with current debugging workflows, with 69% attributing this to the lack of interactive, state-aware tooling.

2.4 Proposed Methodology and Technical Challenges

SolQDebug addresses the two root causes of Solidity's debugging bottleneck—external latency from blockchain round trips, and internal opacity due to storage-based semantics—through a pair of lightweight but complementary techniques.

(1) Eliminating blockchain latency via in-editor interpretation. The traditional debugging workflow requires compilation, deployment, transaction-based state setup, and bytecode tracing—each incurring significant latency. SolQDebug replaces this round trip by performing both parsing and abstract interpretation directly inside the Solidity Editor. To support live editing, we extend the Solidity grammar with interactive parsing rules tailored for isolated statements, expressions, and control-flow blocks. When the developer types or edits code, only the affected region is reparsed using a reduced grammar.

Each parsed statement is inserted into a dynamic control-flow graph (CFG), and abstract interpretation resumes from the edit point. The interpreter uses an interval lattice, assigning each variable a conservative range $[l, h]$ to expose edge conditions (e.g., overflows or failing guards) and to approximate groups of concrete executions that follow the same path. This enables millisecond-scale feedback on code structure and control flow without compilation or chain interaction.

(2) Re-instantiating symbolic state without redeployment. The EVM does not support reverting to a prior state without redeploying the contract or replaying transactions—both of which disrupt iteration. SolQDebug introduces batch annotations as a lightweight mechanism for symbolic state injection. In essence, this reflects a core debugging activity: varying inputs or contract state to observe control-flow outcomes. Rather than reconstructing such conditions through live transactions, developers can write annotations at the top of the function to define initial abstract values. These values are injected before analysis begins and rolled back afterward, ensuring test-case isolation.

This approach brings the debugging workflow closer to the source by making state manipulation explicit and reproducible within the code itself. Developers can explore alternative execution paths by editing annotations alone—without modifying the contract logic or incurring compilation and deployment overhead. It effectively decouples symbolic input configuration from the analysis cycle, while preserving the intuitive debugging process developers already follow.

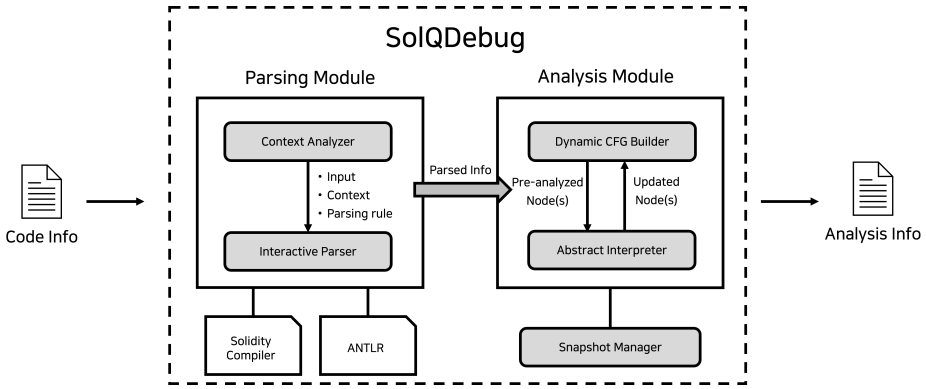


Fig. 2: SolQDebug architecture

3 The design of SolQDebug

3.1 System Architecture

SolQDebug receives incremental edits as its primary input—typically a snippet of Solidity code or an inline debug annotation. Unlike traditional debuggers that require complete programs, SolQDebug is designed to accept fragments ranging from full statements to partial control-flow constructs. These edits include partial Solidity fragments and batch annotations, which are processed in isolation without requiring recompilation or transaction replay. Currently, the system assumes that each line contains at most one statement; compound forms such as `if (...) return;` are not supported. The structure of these inputs is described in Section 3.2; here, we outline the four-stage processing pipeline illustrated in Figure X.

(1) Parsing Module. Each incoming edit first passes through the *Context Analyzer*, which reconstructs a source-level snapshot surrounding the modified lines, determines the enclosing contract or function, and selects the appropriate interactive grammar rule. Subsequently, the *Interactive Parser*, implemented using `?`), applies an extended grammar that incorporates seven additional reduction rules to support isolated Solidity constructs such as expressions, statements, and definitions. A separate rule is dedicated to debug annotations, allowing single-line analysis directives to be parsed as valid units. To ensure syntactic integrity, the reconstructed source is also verified using the Solidity compiler before analysis proceeds. This allows the system to reject malformed fragments early and maintain consistency across the abstract syntax tree and control-flow graph. Debug annotations are parsed as valid syntactic units and forwarded for interpretation; their semantic effects are described in the analysis stage.

(2) Analysis Module. The Analysis Module operates in three coordinated stages. First, each parsed statement is enriched with contextual metadata—including its enclosing contract and function, its semantic role (e.g., declaration or condition), and its static type—before being passed to the *Dynamic CFG Builder*. The CFG Builder maintains an incremental control-flow graph that evolves with each edit: it inserts

a new basic block at the precise edit location, rewires incoming and outgoing control edges, and propagates abstract states along the updated paths. When conditional branches converge, incoming states are merged using the join operator ($\sqcup$); loop headers trigger localized fixpoint computation to ensure convergence. Second, the *Abstract Interpreter* traverses the updated CFG and computes abstract values at each program point. The analysis employs a combination of interval and set domains: numeric types are represented as intervals, addresses as symbolic sets, and composite types (structs, arrays, mappings) are lazily materialized upon field or element access. Third, the *Snapshot Manager* preserves and restores the abstract memory state before and after each batch of debug annotations. This isolation mechanism ensures that consecutive test cases do not interfere with one another, even when they modify shared or local bindings.

(3) Line-Level Output. After interpretation, the system emits a per-statement summary of relevant variable intervals. This includes:

- **Variable declarations:** the initial interval of the declared variable.
- **Assignments:** the updated interval of the left-hand side variable after evaluation.
- **Return statements:** the interval of the returned value or tuple of values.
- **Loops:** intervals for variables that changed during loop execution, computed after fixpoint convergence (loop delta).

All outputs are mapped to source line numbers and displayed directly in the Solidity editor, providing immediate, deployment-free feedback to developers.

3.2 Running Example

To make the architecture concrete, we walk through a small running example that exercises the main components of SolQDebug. The system operates as follows. First, each incremental source fragment is interpreted under abstract semantics to compute interval for the variables it touches. Second, the corresponding expression is stored in a CFG node that is inserted at a semantically valid point, determined from the edit's context and the existing control-flow. Third, when batch annotations are present, the entire function is reinterpreted using the pre-built CFG with the updated abstract state. We illustrate both modes—incremental source edits and batch annotations—using the burn function from Listing 1, and then refer back to the detailed mechanisms in §§3.3–3.5.

3.2.1 Source Code Analysis Example

Table 1 lists the incremental fragments a developer types for the function burn in Listing 1. SolQDebug accepts two kinds of fragments: (i) block fragments such as a function header or an if/else block, and (ii) single statements that end with a semicolon. Most editors auto-insert a closing brace when "{" is typed, so a block fragment arrives as two lines at once (e.g., function ... { and the matching }). As the body is filled, the auto-inserted closing brace is pushed downward. The line numbers shown in the table refer to the listing 1; intermediate edits may temporarily place the closing brace earlier.

Table 1: Incremental inputs for the running example

Step	Lines of Input Fragment	Fragment
1	11--12	`function burn(uint256 amount) public onlyOwner {` `}`
2	12	`uint256 bal = balances[msg.sender];`
3	13	`uint256 delta;`
4	14--15	`if (bal >= amount) {` `}`
5	15	`balances[msg.sender] = bal - amount;`
6	16	`delta = amount;`
7	18--19	`else {` `}`
8	19	`delta = 0;`
9	21	`totalSupply -= delta; // new input`

Figure 3 visualizes the state of the CFG after Steps 1–8 have already been integrated and shows how the new input in Step 9 (totalSupply -= delta;) is analyzed and inserted. For clarity, we focus on the function body in this running example and ignore the modifier referenced in the header; modifiers and their placement are treated in §3.5.

SolQDebug uses the following node semantics and bookkeeping rules:

- **Basic nodes.** A basic node contains a straight-line sequence of statements. As statements are evaluated in order, the node maintains the abstract environment at the end of the node—i.e., the interval for each variable. For later re-analysis (e.g., under batch annotations), the node also records its statement list.
- **Condition nodes.** A condition node stores only the predicate (e.g., `bal >= amount`) and does not update the environment at that point.
- **Branch refinement.** When the true/false successors of a condition are created, the incoming environment is pruned along each edge: the true successor refines intervals under the predicate, while the false successor refines them under its negation. Subsequent statements are analyzed under these pruned environments.

With these rules in place, the arrival of Step 9 proceeds as follows:

(1) Parsing and semantics. The interactive parser recognizes `totalSupply -= delta;` as a single assignment and constructs its abstract transfer function (§3.3).

(2) Finding the insertion point. Using the current edit context (line number and the stack of enclosing constructs) together with the existing CFG, SolQDebug locates the semantically valid insertion point. In this case the context indicates the join after the if/else, not merely the preceding line (§3.5).

(3) Join environment (and localized fixpoint if needed). If the insertion point has multiple predecessors, SolQDebug computes the least upper bound ($\sqcup$, JoinBranches) of their environments to create a join point node. If a loop header is on the path, a localized fixpoint is computed at the header before joining (§3.5).

(4) Evaluation and rewiring. A new basic node is created immediately after the join point node, and the assignment is evaluated in this new node using the joined environment to produce the new interval for `totalSupply`. The outgoing edges from

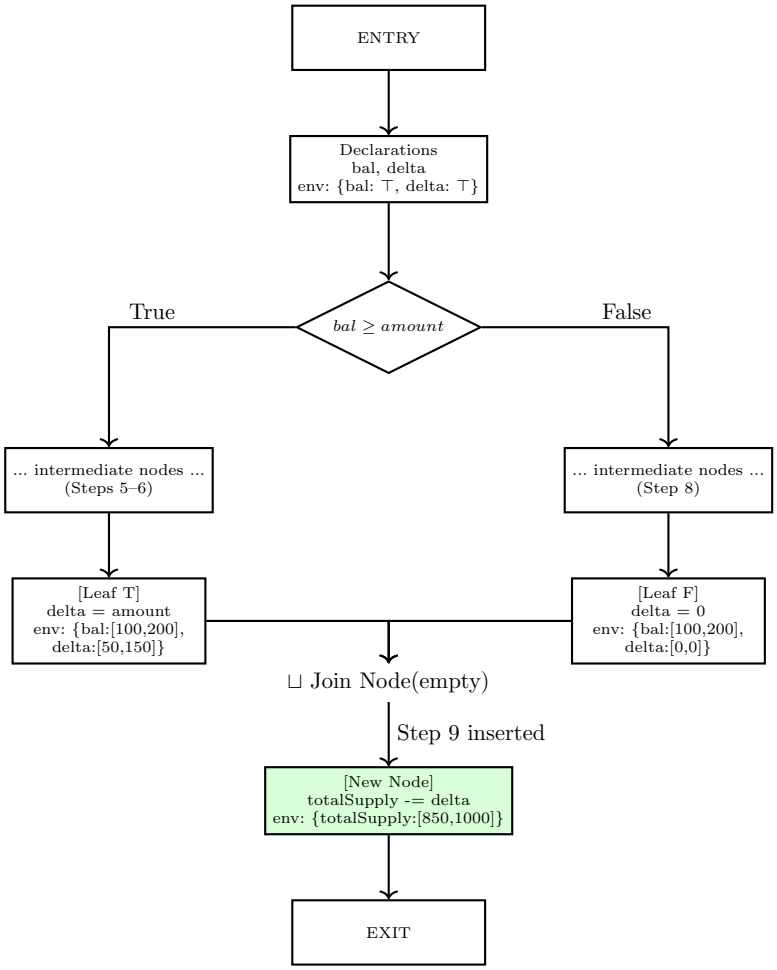


Fig. 3: CFG structure showing Step 9 insertion. Each statement occupies a separate basic node; intermediate nodes along each branch are omitted, showing only the leaf nodes before the join point. The join point node is empty and serves only to merge the environments from both branches

the two branch leaves are rewired to the join point, which flows into the new node and then to the exit.

(5) Reinterpretation from the insertion point. To maintain soundness of the abstract interpretation, SolQDebug reinterprets all nodes reachable from the newly inserted node. Since the insertion changes the incoming environment for subsequent statements, the `reinterpret_From` function propagates the updated environment forward through the CFG, ensuring that all downstream intervals remain valid.

This design allows SolQDebug to reuse all path-local computations accumulated up to the leaves while maintaining semantic correctness at the insertion site. Although the

narrative assumes sequential input, the same procedure applies to out-of-order edits (e.g., adding an else later). The insertion-point search, leaf collection, join/fixpoint handling, and edge rewiring remain unchanged and safely update the existing CFG.

3.2.2 Batch Annotation Analysis Example

Listing 2: Burn function with batch annotations

```
function burn(uint256 amount) public onlyOwner {
    // @Debugging BEGIN
    // @StateVar balances[msg.sender] = [100,200]
    // @LocalVar amount = [50,150]
    // @Debugging END
    uint256 bal = balances[msg.sender];
    uint256 delta;
    if (bal >= amount) {
        balances[msg.sender] = bal - amount;
        delta = amount;
    }
    else {
        delta = 0;
    }
    totalSupply -= delta;
}
```

Batch annotations provide a declarative way for developers to specify initial state and parameters as symbolic (interval) values and to obtain line-level results by reinterpreting the program in a single pass over the already built CFG. In this work, "debugging" refers to the interactive exploration during pre-deployment editing in which the developer varies inputs (and state) to observe branch reachability, guard validity, and value bounds. Consequently, variables that are not given an initial range via annotations remain at the conservative $\top$, which can make results vacuous; this underscores the need for meaningful initialization in debugging. Batch annotations supply this initialization in a consistent and reproducible form.

In the running example, we augment the function `burn` in Listing 1 with the following lines: //@StateVar `balances[msg.sender]` $= [100, 200]$ and //@LocalVar `amount` $= [50, 150]$. We set the initial total supply to $totalSupply = 1000$. This choice makes the condition $bal \geq amount$ partially true, so both the then and else branches are reachable; it thereby exposes how pruning at branches and joining after the conditional affect the resulting bounds.

An annotation block is written between `//@Debugging BEGIN` and `//@Debugging END`, with one directive per line of the form "target L-value $\leftarrow$ abstract value (interval or symbolic)." Targets may be global, state, or local variables, and nested L-values (e.g., `a[i].x`, `balances[addr]`) are allowed. Integers are normalized to intervals respecting their declared bit width; addresses are tracked as symbolic identifiers (set domain); booleans as $\{0, 1\}$.

Given a batch block, SolQDebug executes a lightweight pipeline: (i) parse each line, resolve symbols, and type-check; (ii) snapshot the current abstract memory and

overlay the initial environment with the annotated values; (iii) traverse the existing CFG once from the function entry and perform abstract interpretation; condition nodes record only the predicate and do not immediately change the environment, while the true/false successor blocks refine (prune) their incoming environments under the predicate and its negation; if loops are present, a localized fixpoint is computed at the loop header; and (iv) restore the snapshot to guarantee isolation across runs. This process is coordinated by the Snapshot Manager.

Importantly, batch annotations do not alter the CFG structure. No new nodes are inserted; only the initial environment changes, and the same CFG is reused. Each basic block retains its statement list and the abstract environment at the end of the block (interval per variable), enabling fast reevaluation. The rules for branch pruning, least upper bound (LUB) at joins (JoinBranches), and loop fixpoints are identical to those in the source-code example (§3.2.1). On the pre-built CFG in Figure 3, a batch run proceeds "entry $\rightarrow$ branch pruning $\rightarrow$ join $\rightarrow$ exit" in a single pass.

The concrete effect of the above annotations is as follows. From the statement $bal = \texttt{balances[msg.sender]}$ we obtain

$$bal \in [100, 200], \qquad amount \in [50, 150].$$

The guard $bal \geq amount$ is only partially true, thus both branches are reachable. After pruning, along the true branch the constraint $bal \geq amount$ raises the lower bound of $bal - amount$ to 0, yielding

$$\texttt{balances[msg.sender]} := bal - amount \;\Rightarrow\; [0,\, 200-50] = [0, 150], \qquad \delta := amount \;\Rightarrow\; [50, 150].$$

Along the false branch we only set $\delta := 0$, and `balances[msg.sender]` remains at its annotated initial range $[100, 200]$. At the join we compute

$$\delta \in [50, 150] \sqcup [0, 0] = [0, 150], \qquad \texttt{balances[msg.sender]} \in [0, 150] \sqcup [100, 200] = [0, 200].$$

We then evaluate the assignment to the total supply once in the join environment. With $totalSupply = [1000, 1000]$ initially,

$$totalSupply -= \delta \;\Rightarrow\; [1000, 1000] - [0, 150] = [850, 1000].$$

Thus, by combining branch-specific pruning with an LUB at the join, even a simple interval domain avoids unnecessary blow-up (e.g., the negative region of $bal - amount$ is eliminated on the true path) while conservatively aggregating the effects of both paths.

Containers (arrays, mappings, structs) are kept at $\top$ by default and are concretized on access or when a specific key/field is annotated. In our example, the mapping entry `balances[msg. sender]` is concretized by the annotation, and its effects propagate through the branch body and the join. State variables with explicit initial values, such as *totalSupply*, start from a fixed interval, so a single assignment yields directly interpretable bounds.

In summary, batch annotations standardize the essential debugging act of initial state specification via a simple comment syntax, enabling the developer to explore an intended input range in one shot. SolQDebug reuses the CFG and performs a single-pass reinterpretation, delivering lightweight yet semantically sound results. Formal details and algorithms appear in §3.3 and §3.5.

3.3 Interactive Parser

The standard Solidity parser accepts only whole files (`sourceUnit` → `EOF`) and thus rejects partial fragments produced during editing. SolQDebug's interactive parser extends the official Solidity grammar (defined in `Solidity.g4` (**?**)) with additional entry rules that accept isolated code fragments based on the current editing context. Each fragment is parsed into a syntactically well-formed subtree suitable for incremental analysis without requiring a complete source file.

The interactive parser is built upon ANTLR4 (**?**) and introduces seven specialized entry rules for Solidity program fragments, plus one dedicated rule for batch-annotation blocks. The Context Analyzer determines the appropriate entry rule by inspecting the surrounding source context (e.g., whether the cursor is inside a function body, at top level, or within a struct definition) and selecting the most specific rule. This design allows developers to write and analyze code incrementally—statement by statement—while maintaining full syntactic consistency with the Solidity specification.

Table 2 summarizes the entry rules. Each rule handles specific code fragments encountered during live editing. For instance, when a developer types a function header followed by an auto-inserted closing brace, `interactiveSourceUnit` accepts the skeleton definition. As statements are added inside the body, `interactiveBlockUnit` parses them individually. Two-phase constructs such as enums, structs, `if`/`else`, and `try`/`catch` are handled by dedicated rules that recognize when the first part (e.g., `enum Status` {}) has been declared and the second part (e.g., member items `Pending, Shipped`) is being added incrementally.

Representative example. Consider the `burn` function from Listing 1. When the developer first types the function header, the parser invokes `interactiveSourceUnit` to accept:

```
function burn(uint256 amount) public onlyOwner {
}
```

As statements are added line-by-line inside the body, `interactiveBlockUnit` parses each one:

```
uint256 bal = balances[msg.sender];
uint256 delta;
```

When the developer types an `if` skeleton, `interactiveBlockUnit` matches it as `interactiveIfStatement`:

```
if (bal >= amount) {
}
```

Later, if an `else` branch is added, `interactiveIfElseUnit` recognizes it and attaches it to the preceding `if` using the construct stack maintained by the Context Analyzer.

Table 2: Interactive parser entry rules

Entry Rule	Purpose
`interactiveSourceUnit`	Top-level declarations: functions, contracts, interfaces, libraries, state variables, pragmas, imports
`interactiveEnumUnit`	Enum member items added after the enum shell is defined
`interactiveStructUnit`	Struct member declarations added after the struct shell is defined
`interactiveBlockUnit`	Statements and control-flow skeletons inside function bodies
`interactiveDoWhileUnit`	The `while` tail of a `do{...}` loop
`interactiveIfElseUnit`	`else` or `else if` branches following an `if`
`interactiveCatchClauseUnit`	`catch` clauses following a `try` statement
`debugUnit`	Batch-annotation lines (`@StateVar`, `@LocalVar`)

Batch annotations. The `debugUnit` rule parses annotation lines that specify initial abstract values for state and local variables. These annotations enable developers to configure symbolic input scenarios without deploying contracts. For example:

```
// @Debugging BEGIN
// @StateVar balances[msg.sender] = [100,200]
// @LocalVar amount = [50,150]
// @Debugging END
```

Each directive assigns an interval or symbolic value to a variable. The parser supports nested access patterns (e.g., `balances[msg.sender]`, `user.balance`, `arr[i].field`) and various value types (integer intervals, symbolic addresses, boolean values). Type checking is performed at parse time, and out-of-bounds intervals are clamped to the declared type's valid range with a warning. The Snapshot Manager (§3.4.3) consumes these annotations to overlay the initial abstract environment before function reinterpretation.

The complete grammar specification, including production rules and formal syntax for all entry points, is provided in Appendix A. Implementation details such as context-stack management and entry-rule selection algorithms are also documented there.

3.4 Dynamic CFG Construction

This section explains how we dynamically extend the control-flow graph while a user edits code. We proceed in three steps. First, we construct and splice a CFG fragment for each statement form and rewire only the neighborhood of the current node. Second, we locate the insertion site (the current block) using a successor-first, line-aware selection strategy. Third, we re-interpret only the affected region after splicing to update abstract environments.

3.4.1 Statement-Local, Incremental Construction

We introduce the node kinds that make up the CFG, then show how each major statement is translated into a small CFG fragment and spliced locally. Every edit operates at an INSERTION SITE—the block immediately preceding the new fragment—without restructuring the rest of the graph.

Node kinds.

- BASIC NODE: Holds exactly one statement (e.g., a variable declaration, an assignment, or a function call).
- CONDITION NODE: Represents branching constructs such as `if`, `else if`, `while`, `require`/`assert`, and `try`.
- RETURN NODE: A statement node whose outgoing edge is immediately rewired to the function's unique RETURN EXIT.
- ERROR NODE: The function's unique ERROR EXIT (targets the exceptional path).
- FIXPOINT EVALUATION NODE (ϕ): The loop join used for widening and narrowing.
- LOOP EXIT NODE: The false branch that leaves a loop.

Figure 4 shows a simple statement. The builder creates one BASIC NODE and splices it between the current node and the original successors. Incoming environment is copied from the current node; all outgoing edges of the current node are reattached to the new basic node. We deliberately store *exactly one* statement per basic node so that mid-line insertions become $O(1)$ splices via the editor-to-CFG line map, without scanning or splitting multi-statement blocks.

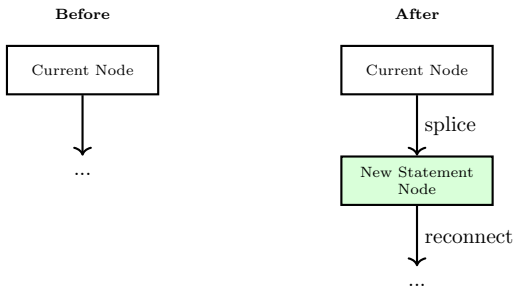


Fig. 4: Simple statement insertion. The builder creates one node and splices it between the current node and the original successors

Figure 5 shows an `if`. The builder inserts a CONDITION NODE for the guard, two BASIC NODES for the true/false arms, and an IF JOIN. Edges: current → condition; condition → true basic (true edge) and → false basic (false edge); both basics → if join; the if join reconnects to the original successors. Environments on the two edges are refined by the truth value of the guard.

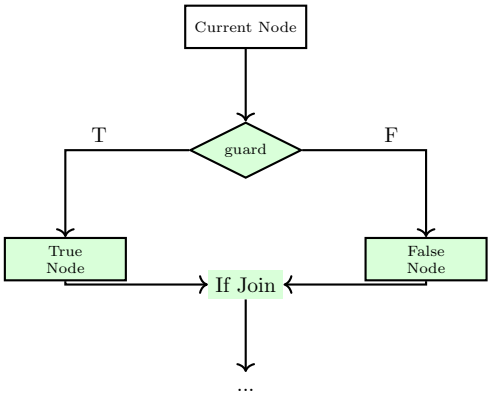


Fig. 5: If statement insertion. The builder creates a CONDITION NODE, two nodes for true/false arms, and an IF JOIN

Figure 6 shows an `else if`. The builder removes the previously created false arm of the nearest preceding `if`/`else if` at the same nesting depth and splices a fragment consisting of a new CONDITION NODE, two BASIC NODES, and an ELSE-IF JOIN. The else-if join is connected to the existing IF JOIN so the overall shape remains a single diamond toward the if join.

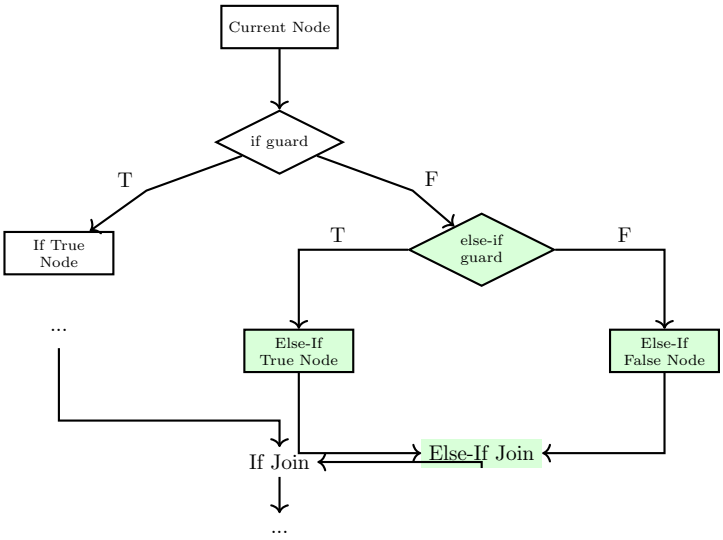


Fig. 6: Else-if statement insertion. The builder replaces the false arm with a new CONDITION NODE, two nodes, and an ELSE-IF JOIN

Figure 7 shows an `else`. No new condition is created; the builder attaches a BASIC NODE to the false branch of the corresponding `if/else if` and connects it to the same IF JOIN as the true branch. The figure assumes a canonical `if/else if/else` chain. For nested patterns (e.g., `if { if {} else {} }`), the else attaches to the false arm of its matching guard according to standard block matching.

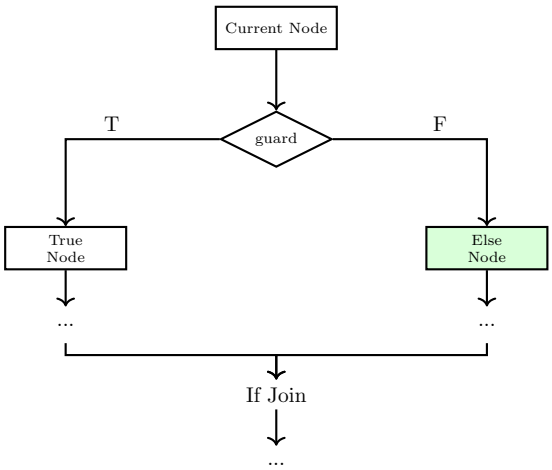


Fig. 7: Else statement insertion. The builder attaches a node to the false branch of the corresponding `if/else if`, connecting to the IF JOIN

Figure 9 shows a `while`. The builder creates a FIXPOINT EVALUATION NODE ϕ, a CONDITION NODE, a true-arm BASIC NODE as the loop-body entry, and a LOOP EXIT NODE (false arm). Rewiring: current $\rightarrow \phi \rightarrow$ condition; condition(true) $\rightarrow$ body; body $\rightarrow \phi$ (back edge); condition(false) $\rightarrow$ loop exit; the loop exit reconnects to the original successors.

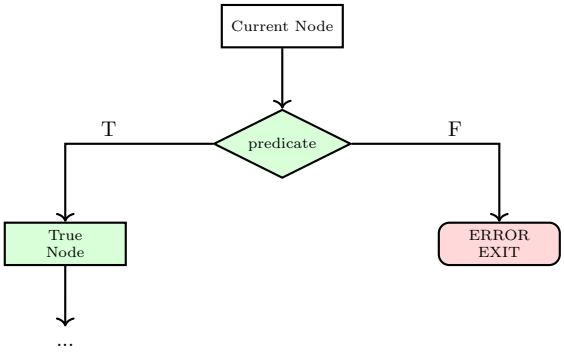


Fig. 8: Require/assert statement insertion. The builder creates a CONDITION NODE with true edge to a node and false edge to the ERROR EXIT

The ϕ node stores both the pre-loop baseline and the running snapshot for widening/narrowing.

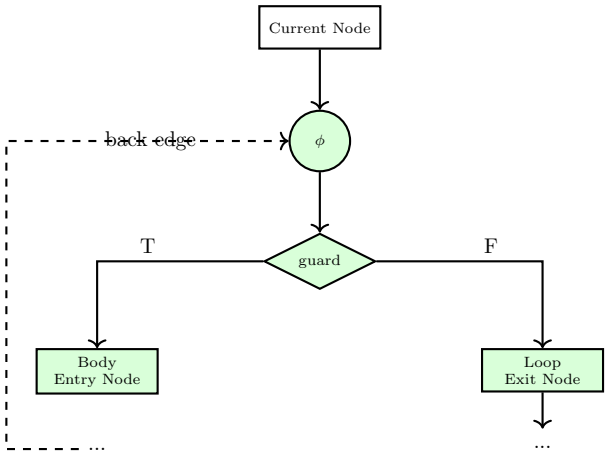


Fig. 9: While loop insertion. The builder creates a FIXPOINT EVALUATION NODE ϕ, a CONDITION NODE, a loop body node, and a LOOP EXIT NODE

Figure 10 shows a `break`. The statement becomes a BASIC NODE whose outgoing edge is redirected to the LOOP EXIT NODE. The loop exit's environment is conservatively joined with the environment at the break site.

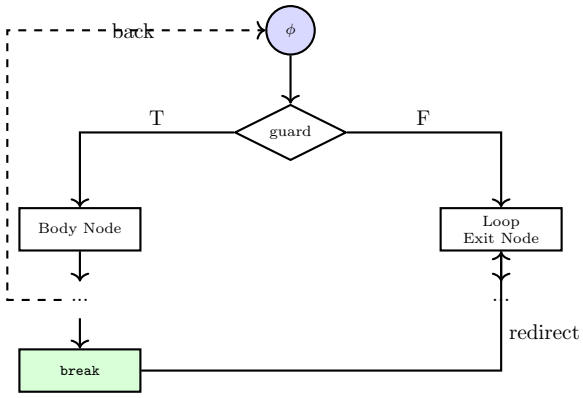


Fig. 10: Break statement insertion. The `break` node's outgoing edge is redirected to the LOOP EXIT NODE

Figure 11 shows a `continue`. The statement becomes a BASIC NODE whose outgoing edge is redirected to the loop's FIXPOINT EVALUATION NODE ϕ. Operationally, this keeps the back-edge shape and joins the current environment into the loop's join state.

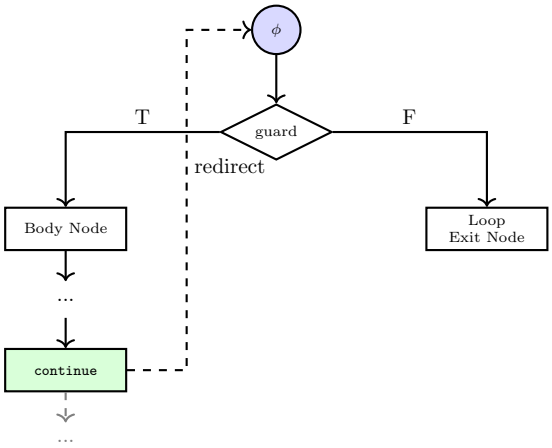


Fig. 11: Continue statement insertion. The `continue` node's outgoing edge is redirected to the loop's FIXPOINT EVALUATION NODE ϕ

Figure 12 shows a `return`. The statement becomes a RETURN NODE and is immediately rewired to the function's unique RETURN EXIT; the return value is recorded there and the original successors of the current node are detached.

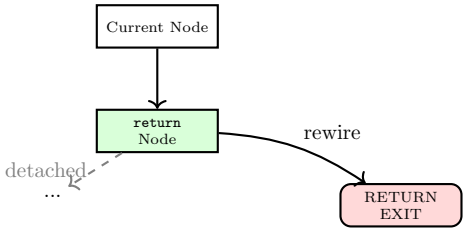


Fig. 12: Return statement insertion. The `return` node is rewired to the function's unique RETURN EXIT

Figure 8 shows `require`/`assert`. The builder inserts a CONDITION NODE for the predicate, makes the true edge point to a BASIC NODE, and connects the false edge directly to the function's ERROR EXIT. The true basic then reconnects to the original successors, forming a one-sided diamond.

Other constructs. Similar patterns apply to the remaining control-flow constructs:

- `for` loops: Handled as a ϕ node and condition like `while`, optionally preceded by an initialization node and followed by an increment node on the back edge.
- `do{} while`: Built in two steps. First, a body pair is created and closed. Later, the trailing `while` line attaches a ϕ, condition, and loop exit, and wires the back edge to the existing body.

- `try{} catch{}`: Represented by a CONDITION NODE tagged as `try` whose true edge goes to the success block and whose false edge is replaced by a catch entry/end pair when a matching `catch` appears.

3.4.2 Line-Aware Successor-First Insertion-Site Selection

We keep a lightweight line–to–node index to make insertion local. Each newly created node is attached to one or two source lines depending on whether the construct is single-line (terminated by ";") or brace-delimited. This index is used only to locate the insertion site; the algorithm below does not mutate the graph.

Line-to-node index mapping. We attach each CFG node to source lines based on the statement type:

- **Sequential statements** (variable declaration, assignment, function call, unary operations): Index the statement block at its source line.
- **Conditional branches** (`if`/`else if`): Index the condition node at the guard line and the join node at the closing brace line.
- **Else branches**: Index the else block at the `else` line and reuse the preceding guard's join node.
- **Loops** (`while`/`for`): Index the condition node at the guard line and the loop-exit node at the closing brace line.
- **Loop control** (`continue`/`break`): Index the statement block at its source line.
- **Terminating statements** (`return`/`revert`): `return` creates a new block; `revert` uses the current block directly.
- **Assertions** (`require`/`assert`): Index the condition node at the statement line.
- **Exception handling** (`try`/`catch`): Index the try condition at the `try` line and catch entry at the `catch` line.

We dispatch insertion-site selection based on the edit context:

- **Branch contexts** (`else`/`else if`/`catch`) use Algorithm 1 to find the preceding condition node by traversing predecessors.
- **Regular statements** use Algorithm 2, which employs a SUCCESSOR-FIRST strategy: locate the earliest CFG node after the edit span and determine the most local predecessor.

Both algorithms never mutate the graph and rely solely on the line–to–node index for efficient lookup.

Algorithm 1: Branch-Context Insertion. For `else`/`else if`/`catch`, we must attach the new branch to a previously created condition node. The algorithm:

- **Line 1–4:** Retrieves CFG nodes at the current line L (or searches backward if L is empty). For `else_if`/`else`, it also identifies the outer join node at L, if present.
- **Line 9–24:** Performs BFS through CFG predecessors to find the matching condition node: for `else_if`/`else`, it looks for a node of type `if` or `else_if`; for `catch`, it looks for type `try`.

Algorithm 1 Branch-Context Insertion-Site Selection (GetBranchContext)

Require: CFG $G = (V, E)$, edit context $ctx \in \{\texttt{else\_if}, \texttt{else}, \texttt{catch}\}$, current line L
Ensure: Condition node $c \in V$ (and optionally outer join $j \in V$ for `else_if`/`else`)

```
1:  N ← NodesAtLine(L)                                  ▷ all CFG nodes indexed at line L
2:  if N = ∅ then
3:      N ← NodesAtPreviousLine(L)                      ▷ search backward if L is empty
4:  end if
5:  j_outer ← ⊥
6:  if ctx ∈ {else_if, else} then
7:      for n ∈ N do
8:          if isJoin(n) then
9:              j_outer ← n  break                      ▷ outer join at current line
10:         end if
11:     end for
12: end if
13: Queue ← N; Visited ← ∅                              ▷ BFS through predecessors
14: while Queue ≠ ∅ do
15:     n ← Dequeue(Queue)
16:     if n ∈ Visited then continue
17:     end if
18:     Visited ← Visited ∪ {n}
19:     if isCond(n) then
20:         τ ← CondType(n)                             ▷ type: if, else_if, try, etc.
21:         if ctx ∈ {else_if, else} and τ ∈ {if, else_if} then
22:             if j_outer = ⊥ then
23:                 j_outer ← OuterJoinFromGraph(n)     ▷ fallback
24:             end if
25:             return (n, j_outer)
26:         else if ctx = catch and τ = try then
27:             return n
28:         end if
29:     end if
30:     for p ∈ Predecessors(n) do
31:         if p ∉ Visited then
32:             Enqueue(Queue, p)
33:         end if
34:     end for
35: end while
36: error "No matching condition node found for context ctx"
```

- **Line 11–12:** When the matching condition is found for `else_if`/`else`, the algorithm returns both the condition node and the outer join (if the outer join was not found at line L, a fallback uses the graph structure).

Algorithm 2 Successor-First Insertion-Site Selection (GetInsertionSite)

Require: CFG $G = (V, E)$, edit span ending at line L
Ensure: Insertion-site node $A \in V$ (no graph mutation here)

$s \leftarrow$ FirstNodeAfter(L) ▷ scan lines $> L$ until the first indexed node
if $s = \perp$ **then**
  $s \leftarrow$ EXIT
end if
$\ell \leftarrow$ LineOf(s)
if isLoopExit(s) **then** ▷ closing a loop
  $N_{prev} \leftarrow$ NodesAtPreviousLine(L) ▷ search backward from L to find previous nodes
  if $N_{prev} \neq \emptyset$ **then**
    $c \leftarrow$ LastCondInNodes(N_{prev}) ▷ check if previous line has a condition
    if $c \neq \perp$ **then**
      return BranchBlock(c, true) ▷ insert in TRUE branch (loop body)
    else
      return Last(N_{prev}) ▷ last node of previous line
    end if
  else
    return FirstPredecessor(s) ▷ fallback
  end if
else if isJoin(s) **then** ▷ closing a selection
  $N_{prev} \leftarrow$ NodesAtPreviousLine(L)
  if $N_{prev} \neq \emptyset$ **then**
    $c \leftarrow$ LastCondInNodes(N_{prev})
    if $c \neq \perp$ **then**
      return BranchBlock(c, true) ▷ default to TRUE branch for `if`
    else
      return Last(N_{prev})
    end if
  else
    return FirstPredecessor(s)
  end if
else ▷ basic successor
  $Pred \leftarrow$ Predecessors(s)
  if $|Pred| = 1$ **then**
    return the unique element of $Pred$
  else
    return NearestByLine($Pred$, L) ▷ choose closest to current line L
  end if
end if

Algorithm 2: Successor-First Insertion. For regular statements, we look ahead to determine the insertion site:

- **Line 1–3:** FirstNodeAfter(L) scans lines strictly larger than L in the line–to–node index and returns the first node; if none is found, we default to `EXIT`.
- **Line 5–14 (loop-exit):** If the successor s is a loop-exit node, we search backward from L to find the previous line's nodes. If the previous line contains a condition node (the loop header), we return its TRUE branch (the loop body entry); otherwise, we return the last node of the previous line.
- **Line 16–23 (join):** If s is a join node (closing a selection), we apply the same backward search. If a condition node is found, we return its TRUE branch (default for `if` constructs); otherwise, return the last previous node.
- **Line 25–28 (basic successor):** If s is a regular basic node, we return its unique predecessor, or choose the predecessor closest to line L if multiple exist.

Helper functions.

- NodesAtLine(L) / NodesAtPreviousLine(L): Return all CFG nodes indexed at line L or the first non-empty line before L.
- FirstNodeAfter(L): Returns the first CFG node indexed at any line $> L$.
- LastCondInNodes(N): Scans node list N in reverse to find the last condition node.
- BranchBlock(c, t): Returns the successor of condition c along the edge labeled with truth value t.
- OuterJoinFromGraph(c): Walks the graph from condition c through its TRUE branch to find the join node.
- NearestByLine(X, ℓ): Returns $\arg\min_{x \in X} |\text{LineOf}(x) - \ell|$.
- Predecessors(s), FirstPredecessor(s): Standard CFG predecessor queries.

3.4.3 Abstract Interpretation for Incremental Analysis

Our system handles two types of edits during interactive debugging, each triggering a different analysis strategy. Debug annotation input follows a batch-and-flush pattern: annotations are accumulated and processed together, culminating in a full interpretation of the entire function CFG from `ENTRY` to `EXIT` (Algorithm 3). This ensures that all annotated inspection points receive freshly computed abstract states. In contrast, source code edits—such as inserting `require`, assignments, or control structures—are processed immediately: dynamic CFG construction (Algorithms 1 and 2) splices the new nodes into the graph, and change-driven reinterpretation (Algorithm 4) propagates updates only along affected paths, providing instant feedback without re-analyzing the entire function. Both strategies invoke the same loop fixpoint subroutine (Algorithm 5) when encountering loop headers.

Algorithm 3 performs initial interpretation when debug annotations are first introduced to a function. It begins with the `ENTRY` node enqueued and propagates abstract environments forward through the entire CFG using a standard worklist iteration. Algorithm 4, in contrast, handles source code edits: the dynamic CFG builder returns one or more seed nodes (never sinks) that mark the insertion points, and the algorithm propagates updates only along forward-reachable paths from these seeds. The choice of seed depends on the statement type: sequential statements (`assignment`, `function call`) seed at the newly inserted block; control-flow constructs (`if`/`while`/`for`) seed

Algorithm 3 Initial Function Interpretation (InterpretFunctionCFG)

Require: CFG $G = (V, E)$ with designated `ENTRY` node
Ensure: All nodes have computed abstract environments

$WL \leftarrow \langle$`ENTRY`$\rangle$; $inQ \leftarrow \{$`ENTRY`$\}$; $Out \leftarrow$ snapshot map
while $WL \neq \langle\rangle$ **do**
  $n \leftarrow WL$.pop(); $inQ \leftarrow inQ \setminus \{n\}$
  $\hat{\sigma}_{in} \leftarrow \bot$ ▷ compute incoming environment from all predecessors
  for all $p \in$ Predecessors(n) **do**
    $\sigma_p \leftarrow (p)$
    if isCond(p) $\wedge$ hasTruthLabel$(p \to n)$ **then**
      $t \leftarrow$ edgeLabel$(p \to n)$
      $\sigma_p \leftarrow$ Refine$(\sigma_p, p.\text{cond}, t)$
      if $\neg$Feasible$(\sigma_p, p.\text{cond}, t)$ **then**
        $\sigma_p \leftarrow \bot$
      end if
    end if
    $\hat{\sigma}_{in} \leftarrow \hat{\sigma}_{in} \sqcup \sigma_p$
  end for
  if isLoopHeader(n) **then**
    $exitNode \leftarrow$ Fixpoint(n) ▷ Algorithm 5
    for all $u \in (exitNode)$ **do**
      if $\neg$isSink(u) $\wedge$ $u \notin inQ$ **then**
        WL.enqueue(u); $inQ \leftarrow inQ \cup \{u\}$
      end if
    end for
    continue
  end if
  $\hat{\sigma}_{out} \leftarrow$ Transfer$(n, \hat{\sigma}_{in})$
  if $\hat{\sigma}_{out} \neq Out[n]$ **then**
    $(n) \leftarrow \hat{\sigma}_{out}$; $Out[n] \leftarrow \hat{\sigma}_{out}$
    for all $u \in (n)$ **do**
      if $\neg$isSink(u) $\wedge$ $u \notin inQ$ **then**
        WL.enqueue(u); $inQ \leftarrow inQ \cup \{u\}$
      end if
    end for
  end if
end while

at their join or loop-exit node to capture all downstream effects; terminating statements (`return`/`revert`) seed at the original successors before rewiring; and assertions (`require`/`assert`) seed at the true-branch successor. Seeds corresponding to sink nodes (`EXIT`, `ERROR`, `RETURN`) are filtered out because they contribute nothing to downstream analysis.

Both algorithms share the same core iteration structure. Each node computes its incoming environment $\hat{\sigma}_{in}$ by joining all predecessor flows. For predecessors that

Algorithm 4 Change-Driven Reinterpretation (ReinterpretFrom)

Require: CFG $G = (V, E)$; seed set S returned by the builder
Ensure: Environments updated along forward-reachable paths from S

$WL \leftarrow \langle\rangle$; $inQ \leftarrow \emptyset$; $Out \leftarrow$ snapshot map
for all $s \in S$ **do** ▷ filter and enqueue all non-sink seeds
  if $\neg$isSink$(s) \wedge s \notin inQ$ **then**
    WL.enqueue(s); $inQ \leftarrow inQ \cup \{s\}$
  end if
end for
while $WL \neq \langle\rangle$ **do**
  $n \leftarrow WL$.pop(); $inQ \leftarrow inQ \setminus \{n\}$
  $\hat{\sigma}_{in} \leftarrow \bot$ ▷ compute incoming environment from all predecessors
  for all $p \in$ Predecessors(n) **do**
    $\sigma_p \leftarrow (p)$
    if isCond$(p) \wedge$ hasTruthLabel$(p \to n)$ **then** ▷ refine by condition
      $t \leftarrow$ edgeLabel$(p \to n)$
      $\sigma_p \leftarrow$ Refine$(\sigma_p, p.\text{cond}, t)$ ▷ apply path constraint
      if $\neg$Feasible$(\sigma_p, p.\text{cond}, t)$ **then**
        $\sigma_p \leftarrow \bot$
      end if ▷ prune infeasible
    end if
    $\hat{\sigma}_{in} \leftarrow \hat{\sigma}_{in} \sqcup \sigma_p$
  end for
  if isLoopHeader(n) **then** ▷ handle loop by local fixpoint
    $exitNode \leftarrow$ Fixpoint(n) ▷ compute fixpoint; returns loop-exit node
    for all $u \in (exitNode)$ **do**
      if $\neg$isSink$(u) \wedge u \notin inQ$ **then**
        WL.enqueue(u); $inQ \leftarrow inQ \cup \{u\}$
      end if
    end for
    continue ▷ skip standard transfer for loop header
  end if
  $\hat{\sigma}_{out} \leftarrow$ Transfer$(n, \hat{\sigma}_{in})$ ▷ apply statement effects
  if $\hat{\sigma}_{out} \neq Out[n]$ **then** ▷ change detected
    $(n) \leftarrow \hat{\sigma}_{out}$; $Out[n] \leftarrow \hat{\sigma}_{out}$
    for all $u \in (n)$ **do**
      if $\neg$isSink$(u) \wedge u \notin inQ$ **then**
        WL.enqueue(u); $inQ \leftarrow inQ \cup \{u\}$
      end if
    end for
  end if
end while

are condition nodes, we apply path-sensitive refinement: the environment is updated according to the condition and the edge's truth label (true or false), and infeasible branches—where the refined environment contradicts the guard—are pruned by setting σ_p to $\bot$. This ensures that only feasible execution paths contribute to the analysis.

Loop headers receive special treatment in both algorithms. When the worklist reaches a loop header, we invoke a dedicated fixpoint procedure (Algorithm 5) that recomputes abstract states for the entire loop body using widening and narrowing. The fixpoint returns the loop-exit node, whose successors are then enqueued for further propagation. This design localizes loop effects: in Algorithm 4, any edit inside a loop body naturally triggers a fresh fixpoint once the header is encountered, without requiring the builder to seed every loop-internal change explicitly.

The change guard mechanism is critical for efficiency. After computing the transfer function's result $\hat{\sigma}_{out}$, we compare it with the previous snapshot $Out[n]$. Only when a change is detected do we update the node's environment, refresh the snapshot, and enqueue non-sink successors. This guarantees termination and avoids redundant work: if downstream nodes remain unaffected, propagation halts. Crucially, it does not miss any updates, because any upstream alteration that modifies a node's input will also alter its output (even for identity transfers on join or condition nodes), thus triggering further propagation.

Adaptive Widening via Iteration Estimation. Traditional fixpoint algorithms apply widening after a fixed number of visits (typically 2) to ensure termination. Algorithm 5 improves precision by computing an *adaptive* threshold τ tailored to each loop's expected iteration count. Line 4 invokes EstimateIterations, which analyzes the loop condition expression to determine τ:

- **Condition structure check:** If the loop condition is a binary comparison with operator $\in \{<, \leq, >, \geq, \neq\}$, proceed; otherwise return the conservative default $\tau = 2$.
- **Operand evaluation:** Evaluate both operands (e.g., loop counter i and bound n) in the pre-loop environment *Start*. If either evaluates to $\bot$ or a non-interval abstract value (e.g., symbolic address, unevaluated state expression), return $\tau = 2$.
- **Iteration count computation:** For interval operands $[\ell_1, u_1]$ and $[\ell_2, u_2]$, compute the maximum iteration count based on the operator. For instance, $i < n$ yields $\tau = u_2 - \ell_1$; $i \leq n$ yields $\tau = u_2 - \ell_1 + 1$.
- **Clamping:** Clamp the result to $[2, 20]$ to balance precision and efficiency. Loops with $\tau \leq 2$ widen immediately; loops with $\tau \in [3, 20]$ iterate precisely up to τ visits before widening; loops exceeding 20 are capped to prevent excessive fixpoint rounds.

This mechanism is *annotation-aware* without requiring special logic: when debug annotations materialize array lengths, mapping sizes, or function parameters, the evaluator naturally computes tighter intervals for condition operands, raising τ and deferring widening. Conversely, unannotated state expressions remain $\bot$, forcing the conservative default and triggering early widening (§**??**, RQ3).

Additionally, line 8 checks CondConverged, an optional early-stopping heuristic that detects when both condition operands stabilize to singleton intervals across

Algorithm 5 Loop Fixpoint at Header

Require: loop header (condition) node h
Ensure: Converged abstract environments at the loop exit and inside the loop

$L \leftarrow$ TraverseLoopNodes(h) ▷ nodes dominated by h and on some back-edge to h
$vis[\cdot] \leftarrow 0$; $In[\cdot], Out[\cdot] \leftarrow \bot$
$Start \leftarrow \bigsqcup\{(p) \mid p \in (h) \setminus L\}$ ▷ pre-loop env (exclude back-edges)
$In[h] \leftarrow Start$
$\tau \leftarrow$ EstimateIterations($h, Start$) ▷ visit threshold for widening

// Widening phase (ascending)
$WL \leftarrow \langle h \rangle$
while $WL \neq \langle\rangle$ **do**
  $n \leftarrow WL$.pop(); $vis[n] \leftarrow vis[n] + 1$
  $\hat{o} \leftarrow$ Transfer($n, In[n]$)
  if IsJoin(n) $\wedge$ $vis[n] > \tau$ **then**
    $\hat{o} \leftarrow$ Widen($Out[n], \hat{o}$)
  else
    $\hat{o} \leftarrow Out[n] \sqcup \hat{o}$
  end if
  if IsJoin(n) $\wedge$ CondConverged(n) **then** ▷ optional early stop
    $Out[n] \leftarrow \hat{o}$; **break**
  end if
  if $\hat{o} \neq Out[n]$ **then**
    $Out[n] \leftarrow \hat{o}$
    for all $s \in (n) \cap L$ **do**
      $In[s] \leftarrow \bigsqcup \{$ Flow($p \to s$) $\mid p \in (s) \cap L \}$ ▷ edge-pruned join
      WL.push(s)
    end for
  end if
end while

// Narrowing phase (descending)
$WL \leftarrow$ any worklist ordering over L
while $WL \neq \langle\rangle$ **do**
  $n \leftarrow WL$.pop()
  $\hat{o} \leftarrow$ Transfer($n, In[n]$)
  if IsJoin(n) **then**
    $\hat{o} \leftarrow$ Narrow($Out[n], \hat{o}$) ▷ at least one round; cap by $k_{\max}$
  end if
  if $\hat{o} \neq Out[n]$ **then**
    $Out[n] \leftarrow \hat{o}$
    for all $s \in (n) \cap L$ **do**
      WL.push(s)
    end for
  end if
end while
return Out ▷ in particular (LoopExit(h)) is now converged

successive iterations, signaling natural convergence and allowing the widening phase to terminate before τ is exhausted.

Abstract Interpretation. At the end of CFG construction, SolQDebug performs abstract interpretation to compute sound over-approximations of variable ranges. We employ interval domains for integer types ($\widehat{\mathbb{Z}}_N, \widehat{\mathbb{U}}_N$), set abstractions for addresses and booleans, and on-demand materialization for composite types (arrays, mappings, structs). Standard widening operators ensure termination in loops. The complete formal semantics—including program syntax, concrete semantics, abstract domains, and abstract transfer functions—are provided in Appendix B.

4 Evaluation

To evaluate how SolQDebug performs in practical debugging scenarios, we organize our study around three research questions:

- **RQ1 – Responsiveness**: How much edit–to–inspect latency does SolQDebug eliminate compared to Remix?
- **RQ2 – Precision Sensitivity to Annotation Structure**: In a common Solidity pattern where inputs are normalized by division, how does the structure of operand intervals—overlapping vs. distinct—impact interval growth?
- **RQ3 – Loops**: Which loop structures lead to loss of precision, and how do symbolic inputs influence the stability of analysis?

4.1 Experimental Setup

We evaluate SolQDebug on a controlled local setup with the following hardware and software configuration:

- **CPU**: 11th Gen Intel® Core™ i7-11390H @ 3.40GHz
- **RAM**: 16.0 GB
- **Operating System**: Windows 10 (64-bit)
- **Implementation Language**: Python

The dataset is derived from DAppSCAN (37), a large-scale real-world benchmark for smart contract analysis comprising 3,345 Solidity files compiled with version `0.8.0` or higher. From this dataset, we selected 30 representative contracts using *complexity-driven stratified sampling* to ensure comprehensive coverage of debugging scenarios encountered in real-world Solidity development.

Our selection criteria focused on three dimensions that directly impact debugging complexity: (1) **computational complexity**— diverse arithmetic operations including percentage calculations (e.g., DapiServer, GreenHouse), time-based vesting logic (e.g., LockupContract, ThorusBond), and conditional capping; (2) **data structure complexity**—complex structs with 5+ fields (10 contracts), nested mappings (e.g., BitBookStake, PercentageFeeModel), dynamic arrays requiring iteration (7 contracts), and mapping-to-struct patterns (12 contracts); and (3) **control flow complexity**— multiple internal function calls (23 contracts), non-trivial loops (5 contracts), nested conditionals with early returns (e.g., PercentageFeeModel with 3-level override hierarchy), and modifier-based access control (4 contracts).

The 30 contracts span multiple DeFi categories including token economics (40%), staking/vesting (30%), DeFi protocols (20%), and utility contracts (10%), with lines of code ranging from 8 to 59 (average: 23 LOC). This sampling strategy ensures coverage of key Solidity idioms—structs, mappings, dynamic arrays, control flow, and arithmetic logic—representing the diverse debugging challenges developers face in practice.

Since Remix IDE lacks built-in automated benchmarking capabilities, we developed `remix_benchmark`, a Selenium-based automation framework that programmatically drives the Remix web interface to measure edit-to-inspect latency. For each test function, `remix_benchmark` automates the full workflow: compilation, contract deployment, state variable initialization via manual storage slot assignment, parameter entry,

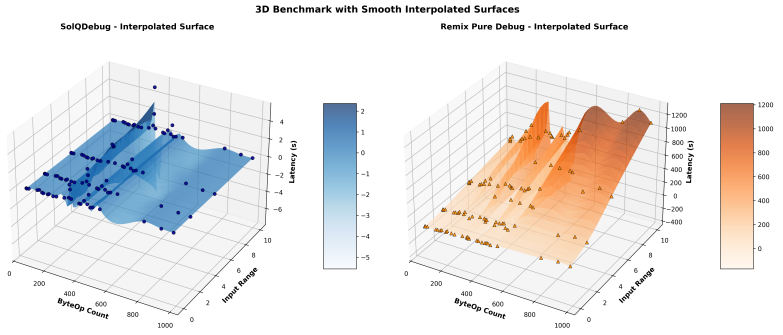


Fig. 13: Edit-to-inspect latency comparison between Remix and SolQDebug across varying test-case widths and execution passes. The x-axis represents the cost estimate, y-axis shows TestCase width (Δ), and z-axis displays latency in seconds. While Remix maintains constant high latency regardless of iteration, SolQDebug demonstrates significantly lower latency that quickly reaches a floor after the initial pass.

transaction execution, and step-through debugging. We measure two latency metrics: *pure debug time*, capturing only the debugger step-through duration, and *total time*, which includes compilation, deployment, and state setup overhead. The difference between these metrics reflects the additional manual effort required in traditional debugging workflows.

Although SolQDebug is designed for interactive use within a Solidity editor, all experiments simulate this behavior in a controlled scripting environment. For each function, we reconstruct a sequence of incremental edits and annotations that mimic realistic developer activity. These fragments are streamed into the interpreter to measure latency and interval growth under reproducible conditions.

4.2 RQ1 - Responsiveness

To evaluate responsiveness, we measure edit-to-inspect latency—defined as the time from a code change to the appearance of updated variable information—under a single contract, single transaction scenario.

We evaluated 30 functions across 4 test-case widths $\Delta \in \{0, 2, 5, 10\}$, yielding 120 total measurements for SolQDebug. For Remix, we measured each function once using `remix_benchmark`, capturing both pure debug time (debugger step-through only) and total time (including compilation, deployment, and state initialization).

For Remix, the pure debug time ranged from 25.1 to 124.6 seconds (median: 53.0 s), reflecting the time required to step through bytecode operations in the debugger. The total time, however, ranged from 71.1 to 168.3 seconds (median: 98.1 s), as it includes approximately 35 seconds for compilation and deployment, plus 0–11.8 seconds for manual state variable initialization (median: 2.9 s). Functions requiring more state slots incur proportionally higher setup overhead, with state initialization time growing linearly with the number of storage variables.

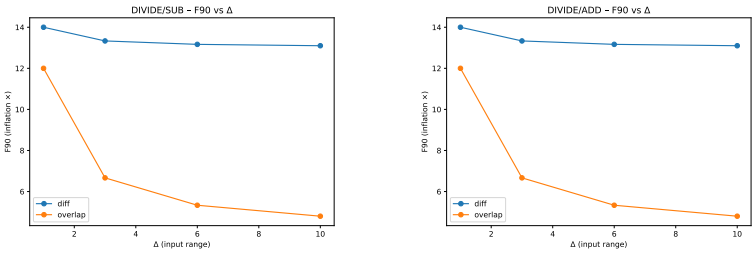


Fig. 14: Interval growth after normalization in `pending` function from `Lock.sol`. Left: original version with subtraction; right: modified version where subtraction is replaced with addition

In contrast, SolQDebug completed analysis in 0.03–5.09 seconds (median: 0.15 s) across all 120 measurements, requiring no compilation, deployment, or state setup. Fig. 13 visualizes this performance gap: Remix pure debug time alone exceeds SolQDebug's total latency by a median factor of ˜350×, while total Remix latency (including setup) exceeds it by ˜650×. This demonstrates that SolQDebug eliminates the compile–deploy–setup cycle entirely, enabling immediate feedback during code editing.

Answer to RQ1: SolQDebug achieves sub-second edit-to-inspect latency (median: 0.15 s), eliminating 350×–650× overhead from Remix's compile–deploy–debug cycle. This enables immediate, interactive feedback during code editing without transaction execution.

4.3 RQ2 - Precision Sensitivity to Annotation Structure

Smart contracts often normalize raw inputs via division—e.g., converting timestamps to time units—before combining the results using addition or subtraction. To isolate the impact of the final arithmetic operator from the shared division step, we analyze two variants of the same control-flow structure: one using addition, the other using subtraction.

Each variant is tested under two annotation styles. In the DIFF style, each operand is assigned a distinct input interval (e.g., $[10, 20]$ and $[30, 40]$). In the OVERLAP style, the intervals are partially aligned (e.g., $[10, 20]$ and $[15, 25]$), such that they share a subrange but are not fully identical. For each combination, we sweep the annotation width $\Delta \in \{1, 3, 6, 10\}$ and report F_{90}, the 90th percentile of the inflation factor $F = \mathit{exit\_width}/\mathit{input\_width}$.

Results in Fig.7 show that interval growth is more sensitive to the structure of input ranges than to the arithmetic operator. DIFF inputs consistently trigger early widening as Δ increases, while OVERLAP inputs maintain tighter bounds even under addition, which typically increases output range.

This suggests that in division-normalized logic, the alignment of operand intervals—whether disjoint or overlapping—has a stronger influence on interval growth than the choice between addition and subtraction. Overlapping inputs consistently

result in smaller output ranges, reducing the degree of over-approximation as input width increases.

Answer to RQ2: In division-normalized arithmetic patterns, the structure of input interval annotations (overlapping vs. disjoint) has a stronger influence on precision than the choice of arithmetic operator (addition vs. subtraction). Overlapping intervals consistently reduce over-approximation by up to 3× compared to disjoint inputs, suggesting developers should align annotation ranges with expected input correlations to improve precision without sacrificing responsiveness.

4.4 RQ3 - Loops

Loop precision in SolQDebug depends critically on the evaluability of the loop condition expression. Recall from Algorithm 5 (line 4) that EstimateIterations analyzes the condition by evaluating both operands in the pre-loop environment *Start* to compute a widening threshold τ. We identify three distinct patterns in our benchmark contracts, each exhibiting different precision characteristics:

Pattern 1: Constant-Bounded Loops. When loop conditions reference only constants or simple arithmetic, EstimateIterations computes precise thresholds without requiring annotations. For instance, `updateUserInfo` in AOC_BEP contains `for (uint256 i = 1; i <= 4; i++)`, where the condition `i <= 4` evaluates to $\tau = 4$. Because the actual loop iterates at most 4 times (including early `break`), widening never triggers, and the interval for `userInfo[account].level` converges precisely to $[1, 4]$. Similarly, `_addActionBuilderAt` in Balancer uses `for (uint8 i = 0; i < additionalCount; i++)`, where `additionalCount` is a locally computed value. When annotations specify the function inputs (e.g., `index = 5`, `currentLength = 2`), the evaluator computes `additionalCount = 4`, yielding $\tau = 4$ and preventing premature widening.

Pattern 2: State-Dependent Bounds Enabled by Annotations. Solidity loops frequently reference dynamic array lengths or mapping sizes in their conditions. Without symbolic input, these expressions evaluate to $\perp$ or remain symbolic, forcing EstimateIterations to return the conservative default $\tau = 2$. For example, `getTotalDeposit` in TimeLockPool iterates `for (uint256 i = 0; i < depositsOf[_account].length; i++)`. The condition operand `depositsOf[_account].length` cannot be statically evaluated, triggering widening after just 2 iterations and causing the accumulator `total` to widen to $[0, 2^{256} - 1]$.

Debug annotations directly address this: by specifying `// @audit depositsOf[_account].length = 10`, the analyzer materializes the array with length 10, allowing EstimateIterations to compute $\tau = 10$. Widening is then deferred until the 10th iteration, enabling precise convergence when the actual loop count matches or falls below the annotated bound. The same mechanism applies to `_removeFromTokens` in AvatarArtMarketplace, where annotating `_tokens.length` allows the threshold to scale proportionally with the input size.

Pattern 3: Data-Dependent Divergence Despite Annotations. Even when annotations provide accurate iteration counts, precision may degrade if the loop

body contains data-dependent branches or updates decoupled from the loop index. `revokeStableMaster` in CORE illustrates this: the loop `for (uint256 i = 0; i < stablecoinListLength - 1; i++)` searches for a specific address in `_stablecoinList` and breaks early upon finding it. Although annotating `_stablecoinList.length` raises τ appropriately, the variable `indexMet` depends on whether the target address exists in the array—a condition orthogonal to the loop index `i`. If the array is populated via annotations but the target is absent, the loop executes the full iteration count, and `indexMet` may widen or remain imprecise due to path merging at the conditional assignment. Conversely, if annotations leave the array empty, the loop terminates immediately with precise state but misses the intended exploration.

This pattern reveals a fundamental tension: annotations that populate data structures enable deeper exploration (activating dormant branches, triggering realistic state transitions), but they also introduce variability that the interval domain cannot track precisely. Such loops require either (i) narrower annotations that materialize only the specific keys or indices the developer intends to inspect, or (ii) acceptance that widening will approximate the full range of behaviors.

Answer to RQ3: Loop precision in SOLQDEBUG is governed by the adaptive widening threshold τ, which is computed by evaluating the loop condition in the pre-loop symbolic environment. Constant-bounded loops converge precisely without annotations. State-dependent loops (e.g., iterating over `array.length`) benefit directly from debug annotations: specifying array sizes or mapping extents allows ESTIMATEITERATIONS to compute accurate thresholds, deferring widening and improving precision proportionally. However, data-dependent loops—where control flow or updates depend on array contents rather than indices—may still diverge under widening despite accurate iteration counts, as the interval domain cannot track element-specific correlations. Developers should annotate iteration bounds for state-dependent loops and use narrower, targeted annotations for data-dependent cases.

5 Discussion

5.1 Why use Abstract Interpretation for Debugging

In this work, we use debugging to mean a developer-led, interactive exploration activity that happens before deployment during code authoring: the developer varies symbolic (interval) inputs and immediately observes branch reachability, guard validity, and value bounds at the source level. This edit-time feedback loop calls for a technique that (1) terminates quickly, (2) explains results in a way developers can inspect, and (3) scales to near-keystroke responsiveness.

We chose abstract interpretation (AI) over symbolic execution and proof-based verification for three reasons:

- **Termination.** AI enforces convergence via widening at loops and joins at merges, avoiding the path explosion common in symbolic execution.

- **Explainability.** Each result is an abstract value in a well-defined lattice. With interval domains, the mapping from inputs to outputs is explicit as ranges, which makes dataflow effects easy to trace and debug at the line level.
- **Responsiveness.** Interval transfer functions are lightweight, enabling millisecond-scale updates that fit the edit cycle. Symbolic engines routinely explore many paths even for small edits, which can break interactivity.

Formal verification provides stronger guarantees, but requires fully specified properties and invariants, which are costly to author during early iterations. SolQDebug is designed to bridge the gap between writing code and running tests or verification—offering immediate, sound, conservative feedback with low annotation overhead.

For debugging, intervals strike a practical balance between precision and speed. They (i) align with developers' mental model of "possible ranges," (ii) expose boundary effects (e.g., overflow thresholds, guard satisfaction regions) without committing to a single concrete input, and (iii) compose predictably through joins and widenings. In our setting, intervals are also a natural surface for annotations: developers can *shape* symbolic inputs (e.g., make them overlapping or disjoint) and directly see how that affects control flow and computed ranges.

AI's precision is conservative by design; edit-time usability depends on giving developers simple levers to steer precision without sacrificing responsiveness. We expose three such levers that proved effective in our study:

- **Annotation structure.** Overlapping operand intervals often bound output ranges more tightly than disjoint ones in division-normalized arithmetic (cf. RQ2). This reduces false alarms with no runtime cost.
- **Annotation width.** Narrower inputs shrink joins and delay widening; developers can start narrow and broaden gradually ("zoom out") to probe stability.
- **Guard-guided narrowing.** Making explicit the intended `require/if` guards in annotations tightens feasible states early and improves precision along the taken branch at negligible cost.

Where stricter precision is essential (e.g., inside data-driven loops), the workflow can temporarily fall back to concrete inputs for local inspection, then return to intervals for broader exploration. This "concrete when needed, symbolic by default" rhythm preserves interactivity while keeping results actionable.

5.2 Evaluation Implication

Traditional debuggers (e.g., Remix, Hardhat Debug) require compile–deploy–execute per iteration, typically taking tens of seconds. In contrast, our interpreter updates in milliseconds (median $\sim$14 ms on the first pass and 5–35 ms on the second), yielding *orders-of-magnitude* lower edit-to-inspect latency. This difference is qualitative: it enables near-keystroke feedback, which changes how developers explore code. Because results are symbolic, a single pass summarizes many concrete executions; developers can see when guards always hold/fail for an interval, when a branch becomes unreachable, or when a value may cross a critical threshold—all without leaving the editor.

In short, SolQDebug complements runtime debuggers by moving fast, informative checks *into* the authoring loop (RQ1).

RQ2 shows that, in division-normalized patterns common in Solidity, *how* intervals are shaped can matter more than *which* arithmetic operator is used. Overlapping inputs systematically produced smaller output ranges than disjoint inputs, delaying or avoiding early widening. When investigating arithmetic joins, start with partially overlapping intervals and widen only as needed; keep operands aligned where normalization is present.

RQ3 demonstrates that loop precision is governed by the adaptive widening threshold τ, which depends on whether the loop condition expression can be evaluated in the pre-loop symbolic environment. Constant-bounded loops (`for (i = 1; i <= 4; ...)`) converge precisely without annotations because EstimateIterations computes exact thresholds. State-dependent loops (`for (i = 0; i < arr.length; ...)`) benefit directly from annotations: specifying `arr.length = 10` allows the analyzer to set $\tau = 10$, deferring widening and enabling natural convergence. However, data-dependent loops—where the loop body contains branches or updates decoupled from the index—may still widen despite accurate iteration counts, as the interval domain cannot track element-specific correlations.

Practical implications: (i) always annotate array lengths, mapping sizes, and iteration bounds for state-dependent loops to raise τ and delay widening; (ii) for data-dependent loops, use narrower annotations that materialize only the specific keys or elements under inspection, reducing the variability introduced by path merging; (iii) when widening becomes unavoidable (e.g., search loops over large collections), accept the over-approximation or switch to concrete inputs for targeted verification.

Overall, these findings suggest a debugging workflow that starts symbolic and broad, then *shapes* annotations to tighten precision where it matters (overlap, narrow, guard-guided), and finally uses concrete spot checks only for stubborn hot spots (e.g., deeply data-dependent loops).

5.3 Limitation

Our current scope and measurements introduce several limitations. First, we focus on single-contract, single-transaction functions. Inter-contract calls, multi-transaction workflows, proxies, and inheritance hierarchies are out of scope in the present implementation. As a result, we have not yet conducted a developer study in larger project settings; the usability and interpretability of edit-time feedback across multi-contract workflows remain unvalidated.

Second, our latency numbers combine interpreter execution time (timed in Python) with an estimate for annotation effort per variable (manual input). This procedure ignores UI-event latency and cursor dynamics, and it assumes a consistent operator for annotation entry. Likewise, our precision metric (F_{90}: 90th percentile of exit-/input-width inflation) captures a salient aspect of interval growth but does not reflect all developer notions of "useful precision." These choices provide a consistent basis for tool-level comparison but may under- or over-estimate end-to-end IDE latency or perceived precision.

We plan to (i) extend the analysis to inter-contract calls and multi-transaction scenarios, (ii) instrument editor events to directly measure human-in-the-loop latency and refine the annotation cost model, and (iii) run a controlled developer study once multi-contract support stabilizes. On the analysis side, loop summarization and selective use of lightweight relational domains (e. g., applied on demand to hot spots) are promising avenues to improve precision while preserving interactivity.

6 Related Works

6.1 Solidity IDEs and Debuggers

Modern Solidity development environments either embed a debugger or integrate external debugging plug-ins. Remix IDE (20) is the most widely used web IDE; it supports syntax highlighting, one-click compilation, and a bytecode-level debugger that lets users step through EVM instructions and inspect stack, memory, and storage. Hardhat (9) is a Node. js–based framework that couples the Solidity compiler with an Ethereum runtime; its Hardhat Debug plug-in attaches a Remix-style debugger to locally broadcast transactions inside Visual Studio Code. Foundry Forge (5) is a command-line toolchain oriented toward fast, reproducible unit testing; the command `forge test` spins up an ephemeral fork, deploys contracts, executes annotated test functions, and enables replay through Forge Debug. Solidity Debugger Pro (27) is a Visual Studio Code extension that performs runtime debugging over concrete transactions and integrates with Hardhat; in practice, many workflows create a small auxiliary contract that calls the target functions so that state changes can be observed step by step.

In short, these debuggers operate on compiled artifacts or post-deployment traces and rely on transaction replay and EVM-level stepping. They do not accept partial, in-flight source fragments nor provide symbolic (interval) input modeling or millisecond edit-time feedback. By contrast, SolQDebug targets pre-deployment authoring, accepts partial fragments and symbolic annotations, and reports line-level effects via abstract interpretation during editing.

6.2 Solidity Vulnerability Detection and Verification

A rich body of work analyzes smart contracts for security issues using four main families of techniques. Static analysis tools reason over source or bytecode without running the contract. Representative systems include rule- or pattern–based analyzers such as Securify and Slither (32; 33), symbolic-execution–assisted detectors like Mythril (35), knowledge-graph–based reasoning such as Solidet (10), and bytecode CFG refinement as in Ethersolve (19). Dynamic testing and fuzzing exercise deployed or locally simulated contracts to uncover faults and security issues: ContractFuzzer mutates ABI-level inputs (12), Echidna brings property-based fuzzing into developer workflows (6), sFuzz adapts scheduling for higher coverage (18), TransRacer finds transaction-ordering races (15), and Ityfuzz leverages snapshotting to decouple executions from chain nondeterminism (23). Formal verification aims to prove safety properties or refute counterexamples at compile time; examples include ZEUS, VeriSmart, and

SmartPulse (13; 24; 31). Finally, AI-based approaches train models to predict vulnerabilities or triage candidates, e. g., via data-flow–aware pretraining, IoT-oriented classifiers, or prompt-tuning for detector adaptation (34; 36; 38).

These approaches have substantially advanced vulnerability detection and property checking for fully written contracts. However, they are not designed to provide interactive, edit-time feedback to developers while code is still under construction. They typically analyze post-compilation artifacts or deployed bytecode and expect complete program units. SolQDebug complements this line of work by focusing on pre-deployment authoring: it accepts partial fragments and symbolic (interval) inputs and produces line-by-line feedback inside the editor.

6.3 Solidity-Specific Abstract Interpretation Frameworks

Abstract interpretation is a well-established framework for static analysis and has been adapted to many programming languages. Two recent studies apply it to Solidity (7; 8). The first uses the Pos domain to construct a theoretical model for taint (information-flow) analysis Halder et al. (7), while the second employs the Difference-Bound Matrix (DBM) domain to generate state invariants and detect re-entrancy vulnerabilities, including the DAO attack (8; 16). However, both approaches operate on fully written contracts and provide no support for line-by-line interpretation or developer interaction within an IDE.

SolQDebug adapts abstract interpretation for an interactive setting. It incrementally updates both the control-flow graph and the abstract state in response to each edit. Developer-supplied annotations serve as a first-class input mechanism, reflecting how debugging often involves varying symbolic inputs. These annotations are internally represented as linear-inequality constraints, and form an integral part of interactive debugging by enabling symbolic reasoning over developer-specified inputs. This design improves interpretability and control within the interval domain by leveraging symbolic constraints, while maintaining keystroke-level responsiveness. As a result, SolQDebug updates variable ranges directly in the Solidity editor, allowing developers to observe how values evolve in response to each edit.

6.4 Interactive Abstract Interpretation for Traditional Languages

In recent years, traditional languages have seen a surge of interest in making abstract interpretation interactive, integrating it directly into IDEs to provide live analysis feedback during editing (1; 4; 21; 29; 30). Stein et al. (29) proposed demanded abstract interpretation, which incrementally rebuilds only the analysis nodes touched by an edit. A follow-up Stein et al. (30) generalized this to procedure summaries, enabling inter-procedural reuse. Erhard et al. (4) extended Goblint with incremental support for multithreaded C, selectively recomputing only genuinely affected facts and maintaining IDE-level responsiveness. Riouak et al. (21) introduced IntraJ, an LSP-integrated analyzer for Java 11 that computes only the AST and data-flow facts needed for the current view, keeping feedback under 100 ms. Chimdyalwar (1) achieved fast yet precise interval analysis on call graphs via one top-down and multiple bottom-up

passes, and later introduced an incremental variant that revisits only the impacted functions.

Unlike these frameworks for C or Java, SolQDebug is designed specifically for Solidity. It supports in-flight code fragments and range annotations as first-class input. It incrementally updates only the current basic block in the CFG while reusing previously computed abstract states. Finally, it combines these with an interval domain guided by developer-supplied annotations, which act as input to represent the exploratory nature of debugging. This architecture enables keystroke-level feedback without requiring recompilation, redeployment, or transaction execution. It bridges the gap between Solidity development and the interactive tooling common in traditional programming environments.

7 Conclusion

We introduced SolQDebug, a source-level interactive debugger for Solidity that provides millisecond feedback without requiring compilation, deployment, or transaction replay. By combining interactive parsing, dynamic control-flow graph updates, and interval domain based abstract interpretation seeded by annotations, SolQDebug enables responsive, line-by-line inspection directly within the Solidity editor. Our evaluation shows that it reduces debugging latency compared to Remix, while enabling actionable feedback in response to symbolic inputs. These results demonstrate that SolQDebug's design effectively bridges the interactivity gap in Solidity debugging and brings the development experience closer to that of modern debugging workflows.

Future work includes extending SolQDebug to inter-contract and multi-transaction contexts, incorporating loop summarization for higher precision, and conducting user studies to assess its practical adoption and usability. We also plan to apply analysis based on the EVM Object Format (EOF) to support inter-contract debugging when source code is unavailable, as Ethereum moves toward structured bytecode formats in upcoming hard forks.

References

Chimdyalwar, B.: Fast and precise interval analysis on industry code. In: 2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW) (2024)

ConsenSys Diligence: Python Solidity Parser. https://github.com/ConsenSysDiligence/python-solidity-parser (2025). Accessed September 2025

Cousot, P., Cousot, R.: 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) (1977)

Erhard, J., et al.: Interactive abstract interpretation: reanalyzing multithreaded C programs for cheap. International Journal on Software Tools for Technology Transfer (2024)

Foundry Forge: https://book.getfoundry.sh/reference/forge/forge/ (2025). Accessed September 2025

Grieco, G., et al.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 557–560 (2020)

Halder, R., et al.: Analyzing information flow in Solidity smart contracts. In: Distributed Computing to Blockchain, pp. 105–123. Academic Press (2023)

Halder, R.: State-based invariant property generation of Solidity smart contracts using abstract interpretation. In: 2024 IEEE International Conference on Blockchain (2024)

Hardhat: https://hardhat.org/ (2025). Accessed September 2025

Hu, T., et al.: Detect defects of Solidity smart contract based on the knowledge graph. IEEE Transactions on Reliability 73(1), 186–202 (2023)

JetBrains: PyCharm. https://www.jetbrains.com/pycharm/ (2025). Accessed September 2025

Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE) , pp. 259–269 (2018)

Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS) (2018)

Llama: https://www.llama.com/ (2025). Accessed September 2025

Ma, C., Song, W., Huang, J.: TransRacer: function dependence-guided transaction race detection for smart contracts. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 947–959 (2023)

Mehar, M.I., et al.: Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. Journal of Cases on Information Technology (2019)

Microsoft Visual Studio: https://visualstudio.microsoft.com/ko/ (2025). Accessed September 2025

Nguyen, T.D., et al.: sFuzz: an efficient adaptive fuzzer for Solidity smart contracts. In: Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE), pp. 778–788 (2020)

Pasqua, M., et al.: Enhancing Ethereum smart-contracts static analysis by computing a precise control-flow graph of Ethereum bytecode. Journal of Systems and Software 200, 111653 (2023)

Remix IDE: https://remix.ethereum.org/ (2025). Accessed September 2025

Riouak, I., et al.: IntraJ: an on-demand framework for intraprocedural Java code analysis. International Journal on Software Tools for Technology Transfer (2024)

Rival, X., Yi, K.: Introduction to Static Analysis: an Abstract Interpretation Perspective (2020)

Shou, C., Tan, S., Sen, K.: Ityfuzz: snapshot-based fuzzer for smart contract. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 322–333 (2023)

So, S., et al.: Verismart: a highly precise safety verifier for Ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1678–1694 (2020)

Solidity Compiler in Python (solcx): https://solcx.readthedocs.io/en/latest/ (2025). Accessed September 2025

Solidity documentation: https://docs.soliditylang.org/en/v0.8.29/ (2025). Accessed September 2025

Solidity Debugger Pro: https://www.soliditydbg.org/ (2025). Accessed September 2025

Solidity Language Grammar Rule of SolQDebug : https://github.com/iwwyou/SolDebug/blob/main/Parser/Solidity.g4 . Accessed September 2025

Stein, B., Chang, B.-Y.E., Sridharan, M.: Demanded abstract interpretation. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) (2021)

Stein, B., Chang, B.-Y.E., Sridharan, M.: Interactive abstract interpretation with demanded summarization. ACM Transactions on Programming Languages and Systems (2024)

Stephens, J., et al.: SmartPulse: automated checking of temporal properties in smart contracts. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 555–571 (2021)

Tsankov, P., et al.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 67–82 (2018)

Tsankov, P., et al.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8–15 (2019)

Wu, H., et al.: Peculiar: smart contract vulnerability detection based on crucial data-flow graph and pre-training techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), pp. 378–389 (2021)

Yao, Y., et al.: An improved vulnerability detection system of smart contracts based on symbolic execution. In: 2022 IEEE International Conference on Big Data (Big Data), pp. 3225–3234 (2022)

Yu, L., et al.: PSCVFinder: a prompt-tuning based framework for smart contract vulnerability detection. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), pp. 556–567 (2023)

Zheng, Z., et al.: Dappscan: building large-scale datasets for smart contract weaknesses in dApp projects. IEEE Transactions on Software Engineering (2024)

Zhou, Q., et al.: Vulnerability analysis of smart contract for blockchain-based IoT applications: a machine learning approach. IEEE Internet of Things Journal 9(24), 24695–24707 (2022)

Zou, W., et al.: Smart contract development: challenges and opportunities. IEEE Transactions on Software Engineering (2019)

A Interactive Parser Grammar Specification

This appendix provides the complete grammar specification for SolQDebug's interactive parser.

A.1 Entry Rules for Solidity Program Fragments

A.1.1 Rule 1: `interactiveSourceUnit`

Purpose. Accepts top-level declarations: functions, contracts, interfaces, libraries, state variables, pragmas, and imports.
Grammar:

```
interactiveSourceUnit
  : (interactiveStateVariableElement | interactiveFunctionElement
    | interfaceDefinition | libraryDefinition | contractDefinition
    | pragmaDirective | importDirective)* EOF ;
```

A.1.2 Rule 2: `interactiveEnumUnit`

Purpose. Accepts enum member items added after the enum shell.
Grammar:

```
interactiveEnumUnit : (interactiveEnumItems)* EOF;
interactiveEnumItems : identifier (',' identifier)*;
```

A.1.3 Rule 3: `interactiveStructUnit`

Purpose. Accepts struct member declarations added after the struct shell.
Grammar:

```
interactiveStructUnit : (structMember)* EOF;
structMember : typeName identifier ';' ;
```

A.1.4 Rule 4: `interactiveBlockUnit`

Purpose. Accepts statements and control-flow skeletons inside function bodies.
Grammar:

```
interactiveBlockUnit : (interactiveBlockItem)* EOF;
interactiveBlockItem : interactiveStatement | uncheckedBlock;
```

A.1.5 Rule 5: `interactiveDoWhileUnit`

Purpose. Accepts the `while` tail of a `do{...}` loop.
Grammar:

```
interactiveDoWhileUnit : (interactiveDoWhileWhileStatement)* EOF;
```

```
interactiveDoWhileWhileStatement : 'while' '(' expression ')' ';' ;
```

A.1.6 Rule 6: `interactiveIfElseUnit`

Purpose. Accepts `else` or `else if` branches.
Grammar:

```
interactiveIfElseUnit : (interactiveElseStatement)* EOF;
interactiveElseStatement : 'else' (interactiveIfStatement | '{' '}') ;
```

A.1.7 Rule 7: `interactiveCatchClauseUnit`

Purpose. Accepts `catch` clauses following a `try`.
Grammar:

```
interactiveCatchClauseUnit : (interactiveCatchClause)* EOF;
interactiveCatchClause : 'catch' (identifier? '(' parameterList ')')? '{' '}' ;
```

A.2 Entry Rule for Debugging Annotations

A.2.1 `debugUnit`

Purpose. Parses batch-annotation lines that specify initial abstract values for variables.
Annotation types:

- `@GlobalVar`: Assigns values to global variables (e.g., `msg.sender`, `block.timestamp`)
- `@StateVar`: Assigns values to contract state variables
- `@LocalVar`: Assigns values to function parameters and local variables

Grammar:

```
debugUnit : (debugGlobalVar | debugStateVar | debugLocalVar)* EOF;
debugGlobalVar : '//' '@GlobalVar' identifier ('.' identifier)? '=' globalValue ';' ;
debugStateVar : '//' '@StateVar' lvalue '=' value ';' ;
debugLocalVar : '//' '@LocalVar' lvalue '=' value ';' ;
```

Supported L-value patterns: Simple variables, array/mapping access (`arr[i]`, `map[key]`), struct fields (`s.field`), and nested combinations.
Value specification: Integer intervals `[l,u]`, symbolic addresses `symbolicAddress n`, boolean values, and symbolic placeholders.

B Abstract Domain and Formal Semantics

This appendix presents the abstract domain definitions and formal semantics used by SolQDebug's abstract interpreter. The framework is based on interval analysis for

numeric types, set domains for addresses, and lazy materialization for composite data structures.

B.1 Abstract Domain

Atomic abstract values:

- **Unsigned integers:** $\widehat{\mathbb{U}}_N = \{[\ell, u] \mid 0 \le \ell \le u \le 2^N{-}1\} \cup \{\bot, \top_N\}$
- **Signed integers:** $\widehat{\mathbb{Z}}_N = \{[\ell, u] \mid -2^{N-1} \le \ell \le u \le 2^{N-1}{-}1\} \cup \{\bot, \top_N^{\pm}\}$
- **Booleans:** $\widehat{\mathbb{B}} = \{\bot, \widehat{\mathsf{false}}, \widehat{\mathsf{true}}, \top\}$
- **Addresses:** $\widehat{\mathbb{A}} = \wp_{\le K}(\mathsf{AddrID}) \cup \{\top\}$ (set domain with cap $K = 8$)
- **Bytes:** $\widehat{\mathbb{BY}}_K = \{\bot, \top_K\}$ (symbolic/opaque)
- **Enums:** $\widehat{\mathsf{Enum}}(E) = \{[\ell, u] \mid 0 \le \ell \le u \le |E| - 1\} \cup \{\bot, \top_E\}$

Composite values:

- **Structs:** $\widehat{\mathsf{Struct}}(C) = \prod_{f \in \mathsf{fields}(C)} \widehat{\mathsf{Val}}_f$ (pointwise order)
- **Arrays:** $\widehat{\mathsf{Arr}}(\tau) = (\hat{\ell}, \hat{d}, M)$ where $\hat{\ell} \in \widehat{\mathbb{U}}_{256}$ is length, $\hat{d}$ is default element, $M : \mathbb{N}_{\mathrm{fin}} \rightharpoonup \widehat{\tau}$ stores observed indices
- **Mappings:** $\widehat{\mathsf{Map}}(\kappa \Rightarrow \tau) = (\hat{d}, M)$ with default $\hat{d}$ and finite map M for observed keys

Order, join, and meet: For intervals: $[\ell_1, u_1] \sqsubseteq [\ell_2, u_2] \iff \ell_2 \le \ell_1 \wedge u_1 \le u_2$, $[\ell_1, u_1] \sqcup [\ell_2, u_2] = [\min(\ell_1, \ell_2), \max(u_1, u_2)]$, $[\ell_1, u_1] \sqcap [\ell_2, u_2] = [\max(\ell_1, \ell_2), \min(u_1, u_2)]$ if non-empty, else $\bot$.

Widening ∇ and narrowing Δ follow standard interval analysis patterns. For address sets: $S_1 \sqcup S_2 = S_1 \cup S_2$ if $|S_1 \cup S_2| \le K$, else $\top$.

B.2 Concrete Semantics (Denotational)

Let stores be $\sigma : \mathsf{Var} \rightharpoonup \mathsf{CVal}$. L-value resolution $\mathrm{loc}_\sigma(lv) = \ell$ and write $\mathrm{write}(\sigma, \ell, v)$ update the store. Expressions are pure: $[\![e]\!]_\sigma \in \mathsf{Val}$.

Outcome domain:

$$\mathsf{Res} \ ::= \ \mathsf{Norm}(\sigma) \ \mid \ \mathsf{Ret}(v, \sigma) \ \mid \ \mathsf{Abort}$$

with sequencing

$$\begin{aligned} \mathsf{Norm}(\sigma) \ &\triangleright \ K := K(\sigma), \\ \mathsf{Ret}(v, \sigma) \ &\triangleright \ K := \mathsf{Ret}(v, \sigma), \\ \mathsf{Abort} \ &\triangleright \ K := \mathsf{Abort}. \end{aligned}$$

Array/mapping materialization: $\mathrm{loc}_\sigma(a[i])$ extends a up to i with defaults if needed; $\mathrm{loc}_\sigma(m[k])$ creates $m[k]$ lazily if absent.

Table 3: Concrete denotational semantics (statements)

Statement	**Meaning**
skip	$[\![\mathsf{skip}]\!](\sigma) = \mathsf{Norm}(\sigma)$
$s_1; s_2$	$[\![s_1; s_2]\!](\sigma) = \big([\![s_1]\!](\sigma)\big) \triangleright (\lambda\sigma'.\, [\![s_2]\!](\sigma'))$
$\tau\ x;$	$[\![\tau\ x;]\!](\sigma) = \mathsf{Norm}\big(\sigma[x \mapsto \mathrm{zero}_\tau]\big)$
$\tau\ x = e;$	$[\![\tau\ x = e;]\!](\sigma) = \mathsf{Norm}\big(\sigma[x \mapsto [\![e]\!]_\sigma]\big)$
$lv := e$	$[\![lv := e]\!](\sigma) = \mathsf{Norm}\big(\mathrm{write}(\sigma,\, \mathrm{loc}_\sigma(lv),\, [\![e]\!]_\sigma)\big)$
$\mathsf{delete}\ lv$	$[\![\mathsf{delete}\ lv]\!](\sigma) = \mathsf{Norm}\big(\mathrm{write}(\sigma,\, \mathrm{loc}_\sigma(lv),\, \mathrm{zero}_{\tau(lv)})\big)$
$\mathsf{if}\ p\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f$	$[\![\cdot]\!](\sigma) = \begin{cases} [\![s_t]\!](\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{true}, \\ [\![s_f]\!](\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{false} \end{cases}$
$\mathsf{while}\ p\ \mathsf{do}\ s$	$F(H)(\sigma) = \begin{cases} \big([\![s]\!](\sigma)\big) \triangleright H & \text{if } [\![p]\!]_\sigma = \mathsf{true}, \\ \mathsf{Norm}(\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{false} \end{cases}$; $[\![\mathsf{while}\ p\ \mathsf{do}\ s]\!] = \mathrm{lfp}(F)$
$\mathsf{return}\ e$	$[\![\mathsf{return}\ e]\!](\sigma) = \mathsf{Ret}([\![e]\!]_\sigma,\, \sigma)$
$\mathsf{assert}(p)$, $\mathsf{require}(p)$	$[\![\cdot]\!](\sigma) = \begin{cases} \mathsf{Norm}(\sigma) & \text{if } [\![p]\!]_\sigma = \mathsf{true}, \\ \mathsf{Abort} & \text{if } [\![p]\!]_\sigma = \mathsf{false} \end{cases}$
$\mathsf{revert}(\cdots)$	$[\![\mathsf{revert}(\cdots)]\!](\sigma) = \mathsf{Abort}$
$\mathsf{call}(\overline{e})$	Internal: parameter binding; external: unspecified

B.3 Abstract Semantics (Denotational)

Let $\hat{\sigma} : \mathsf{Var} \rightharpoonup \widehat{\mathsf{CVal}}$ be the abstract store. Expressions evaluate to $[\![e]\!]^\sharp_{\hat{\sigma}} \in \widehat{\mathsf{Val}}$.
Abstract outcomes:

$$\widehat{\mathsf{Res}} \ ::= \ \widehat{\mathsf{Norm}}(\hat{\sigma}) \ \mid \ \widehat{\mathsf{Ret}}(\hat{v}, \hat{\sigma}) \ \mid \ \widehat{\mathsf{Abort}},$$

with sequencing

$$\begin{aligned} \widehat{\mathsf{Norm}}(\hat{\sigma}) \ &\triangleright^\sharp \ K := K(\hat{\sigma}), \\ \widehat{\mathsf{Ret}}(\hat{v}, \hat{\sigma}) \ &\triangleright^\sharp \ K := \widehat{\mathsf{Ret}}(\hat{v}, \hat{\sigma}), \\ \widehat{\mathsf{Abort}} \ &\triangleright^\sharp \ K := \widehat{\mathsf{Abort}}. \end{aligned}$$

Auxiliary functions:

- $\mathrm{refine}(\hat{\sigma}, p, b)$: narrows operands of p by interval meets
- $\widehat{\mathrm{write}}(\hat{\sigma}, lv, \hat{v})$: strong update if singleton index/key, weak update otherwise
- $\mathrm{joinRes}(r_1, r_2)$: componentwise join of abstract outcomes

Expression semantics: Arithmetic $(+, -, *, /, \%)$: interval arithmetic with wrapping; comparisons $(<, \leq, =, \neq, \geq, >)$: abstract booleans; logical $(\wedge, \vee, \neg)$: three-valued logic.
Array/mapping access: Singleton index/key: strong update; range/non-singleton: join of materialized cells.

Table 4: Abstract denotational semantics (statements)

Statement	**Meaning**
skip	$[\![\mathsf{skip}]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}(\hat{\sigma})$
$s_1; s_2$	$[\![s_1; s_2]\!]^\sharp(\hat{\sigma}) = \big([\![s_1]\!]^\sharp(\hat{\sigma})\big) \ \triangleright^\sharp \ (\lambda\hat{\sigma}'.\, [\![s_2]\!]^\sharp(\hat{\sigma}'))$
$\tau\ x;$	$[\![\tau\ x;]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}\big(\hat{\sigma}[x \mapsto \hat{\mathrm{init}}(\tau)]\big)$
$\tau\ x = e;$	$[\![\tau\ x = e;]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}\big(\hat{\sigma}[x \mapsto \alpha_\tau([\![e]\!]^\sharp_{\hat{\sigma}})]\big)$
$lv := e$	$[\![lv := e]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}\big(\widehat{\mathrm{write}}(\hat{\sigma},\ lv,\ [\![e]\!]^\sharp_{\hat{\sigma}})\big)$
$\mathsf{delete}\ lv$	$[\![\mathsf{delete}\ lv]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Norm}}\big(\widehat{\mathrm{write}}(\hat{\sigma},\ lv,\ \hat{\mathrm{zero}}_{\tau(lv)})\big)$
$\mathsf{if}\ p\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f$	$\hat{\sigma}_t = \mathrm{refine}(\hat{\sigma}, p, \mathsf{true})$, $\hat{\sigma}_f = \mathrm{refine}(\hat{\sigma}, p, \mathsf{false})$; $[\![\cdot]\!]^\sharp(\hat{\sigma}) = \mathrm{joinRes}\big([\![s_t]\!]^\sharp(\hat{\sigma}_t),\ [\![s_f]\!]^\sharp(\hat{\sigma}_f)\big)$
$\mathsf{while}\ p\ \mathsf{do}\ s$	$G^\sharp(H)(\hat{\sigma}) = \mathrm{joinRes}\big([\![s]\!]^\sharp(\mathrm{refine}(\hat{\sigma}, p, \mathsf{true}))\ \triangleright^\sharp\ H,\ \widehat{\mathsf{Norm}}(\mathrm{refine}(\hat{\sigma}, p, \mathsf{false}))\big)$; $[\![\mathsf{while}\ p\ \mathsf{do}\ s]\!]^\sharp = \mathrm{lfp}^\nabla(G^\sharp) \triangle \mathrm{narrow}^k$
$\mathsf{return}\ e$	$[\![\mathsf{return}\ e]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Ret}}([\![e]\!]^\sharp_{\hat{\sigma}},\ \hat{\sigma})$
$\mathsf{assert}(p)$, $\mathsf{require}(p)$	$\widehat{\mathsf{Norm}}(\mathrm{refine}(\hat{\sigma}, p, \mathsf{true}))$ if p must-hold; $\widehat{\mathsf{Abort}}$ if p must-fail; joinRes otherwise
$\mathsf{revert}(\cdots)$	$[\![\mathsf{revert}(\cdots)]\!]^\sharp(\hat{\sigma}) = \widehat{\mathsf{Abort}}$
$\mathsf{call}(\overline{e})$	Internal: parameter binding; external: havoc footprint or $\widehat{\mathsf{Abort}}$