Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

# svlang and svlang-lint: A Pure-Rust IEEE 1800-2023 SystemVerilog Compiler Front End and Semantic Static Analyzer for Hardware Design and Verification

**Mayur Kubavat**

mkubavat01@gmail.com

Independent Researcher

**Research Article**

**Keywords:**

**Posted Date:** November 25th, 2026

**DOI:** https://doi.org/10.21203/rs.3.rs-11813840/v1

**License:** This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

**Additional Declarations:** No competing interests reported.

# svlang and svlang-lint: A Pure-Rust IEEE 1800-2023 SystemVerilog Compiler Front End and Semantic Static Analyzer for Hardware Design and Verification

**Mayur Kubavat** [1,*]

[1] Independent Researcher, Mountain View, CA, USA

[*] Corresponding author: Mayur Kubavat. E-mail: mkubavat01@gmail.com; ORCID: 0009-0009-1087-8558

**Abstract** **Purpose:** Open-source static analysis for SystemVerilog (IEEE 1800-2023) and the Universal Verification Methodology (UVM, IEEE 1800.2-2020) is split between two incomplete tool classes. Concrete-syntax-tree (CST) linters check unelaborated tokens without resolving types, class specializations, or component hierarchies, while open-source RTL simulators elaborate synthesizable logic but omit large parts of object-oriented verification, constrained randomization, and UVM semantics. **Methods:** This paper presents `svlang`, a SystemVerilog (IEEE 1800-2023) front end and word-level RTL/SVA lowering library written in safe Rust (143,840 lines across 20 workspace crates, with zero external Cargo dependencies in the 12-crate compiler and lowering core). Built on top of `svlang`, `svlang-lint` provides a parallel semantic static analyzer, a CST repair engine, and a language server (`dv-lsp`). Ten differential parity gates verify preprocessing, lossless parsing, four-state constant evaluation, IEEE 1800 §§11.6/11.8 context-determined expression typing, UVM class specialization, and netlist lowering against the C++ `Slang` v12.1 reference compiler. On this elaborated model, `svlang-lint` executes 433 rule implementations (461 catalog entries across 20 categories) covering polynomial-time constraint interval and difference-bound cycle checks, class-lineage UVM objection and TLM-to-covergroup flow checks, multi-clock CDC and FSM reachability checks, and provenance-guarded CST autofixes across 38 rule modules. **Results:** `svlang` passes 1,024/1,024 (100.0%) applicable CHIPS Alliance `sv-tests` compliance cases with zero false accepts or rejects, matches `Slang` on 99,712 typed expressions across 904 compilations and 172/172 UVM 2020.3.1 class specializations, and lowers 1,081 standalone RTL top modules (68.7% under strict defaults; 86.5% when time-zero check flags are relaxed) across 16 open-source silicon repositories at 48,834 lines/s. On 415 paired (`bad.sv, good.sv`) defect fixtures, `Verible`, `Verilator`, and `Icarus Verilog` together discriminate 122/415 rules (29.4% union), missing 293/415 (70.6%) verification, constraint, UVM, and CDC defects. Semantic resolution removes all `ERROR` false positives on OpenTitan's shared DV libraries (170 → 0) and Accellera `uvm-core` (8 → 0), produces 30/30 (100.0%) compile-clean CST rewrites on active autofix fixtures (29/30 single-run fixed-point convergence), lints the 86,476-line `uvm-core` library in 1.43 s (23.7× warm replay speedup via `.svlang_lint_cache`), and exposes empty coverage-collector stubs in `core-v-verif`. **Conclusion:** Keeping macro token provenance alongside full IEEE 1800-2023 elaboration in Rust makes it practical to check RTL, UVM hierarchies, and constraints together and apply safe source fixes at interactive speeds.

**Keywords:** SystemVerilog · IEEE 1800-2023 · Universal Verification Methodology (UVM) · Static analysis · Compiler front end · Automated program repair · Design verification · Rust

## 1 Introduction

As digital integrated circuits grow in logic density and protocol complexity, functional verification accounts for the majority of engineering effort in system-on-chip (SoC) development. Open-source silicon projects such as OpenTitan [1], Caliptra [2], and OpenHW CORE-V [3] write their RTL designs in IEEE 1800-2023 SystemVerilog [4] and verify them with layered, reusable testbenches built on the IEEE 1800.2-2020 Universal Verification Methodology (UVM) [5]. In software engineering, shared compiler front ends routinely power static bug checkers, source refactoring tools, and Language Server Protocol (LSP) [6] editors. SystemVerilog resists this treatment because one preprocessor and type system cover two very different domains: cycle-accurate register-transfer level (RTL) hardware with SystemVerilog Assertions (SVA), and object-oriented verification environments with parameterized classes, factory overrides, constrained-random stimulus, functional covergroups, and Register Abstraction Layer (RAL) models.

In commercial ASIC verification, teams rely on proprietary static analyzers such as AMIQ Verissimo, Synopsys VC SpyGlass, Siemens Questa Lint, or RealIntent AscentLint to check RTL and UVM code before simulation. In open-source Electronic Design Automation (EDA), existing tools split along the design-versus-testbench boundary:

1. **Syntax-tree linters** such as CHIPS Alliance `Verible` [7] and Hatta's `dalance/svlint` (built on `sv-parser`) [8] work directly on unelaborated concrete syntax trees (CSTs). They are fast and effective for formatting and naming style, yet they do not expand preprocessor macros with dual invocation and definition coordinates, fold parameter-dependent widths, separate `typedef` aliases from class hierarchies, or check TLM and configuration wiring across UVM components.

2. **Open-source simulator and synthesis front ends** such as `Verilator` [9], `Icarus Verilog` [10], and `Yosys` [11] elaborate synthesizable RTL thoroughly, but omit or only partially model verification constructs such as parameterized UVM class specializations, `uvm_config_db` type matching, RAL address maps, and covergroup sampling. Simulator front ends also discard whitespace trivia and macro-expansion boundaries during AST lowering, preventing byte-accurate source autofixes (`--fix`) and low-latency editor overlays.

Full C++ SystemVerilog front ends such as Popoloski's

Slang [12] and CHIPS Alliance Surelog/UHDM [13] parse the complete language, yet static bug finding and source repair add two practical requirements of their own. First, UVM testbenches use deeply nested macros (`uvm_component_utils, `DV_CHECK, `downcast). When an analyzer conflates the macro *invocation site* with the *written token site*, diagnostics point inside library headers and automated edits corrupt macro bodies. Second, as Bessey et al. [14] documented in industrial static analysis, syntactic shortcuts trigger false alarms on clean idioms until engineers turn rules off.

Listing 1: Two SystemVerilog/UVM cases where syntax-only checks fail in opposite ways: (a) a vector cast misread as an unsafe OOP downcast when declarations are not resolved, and (b) a transitive constructor virtual call and contradicted constraint invisible to single-method or RTL-only tools.

```
// (a) OpenTitan idiom: typedef bit [63:0] uvm_reg_data_t;
// Name check sees 'uvm_*'(...) and flags an OOP downcast;
// symbol lookup shows uvm_reg_data_t is a 64-bit bit vector.
uvm_reg_data_t raw = uvm_reg_data_t'(csr_val);

// (b) new() calls non-virtual init_cfg(), which calls virtual
// build_map() before subclass fields initialize (SV-OOP-017);
// hard constraint c_lim has an empty 4-bit range (SV-RND-030).
class base_seq extends uvm_sequence;
  rand bit [3:0] burst_len;
  constraint c_lim { burst_len > 4'd12; burst_len < 4'd8; }
  function new(string name = "base_seq");
    super.new(name);
    init_cfg(); // non-virtual helper -> virtual build_map()
  endfunction
  function void init_cfg(); build_map(); endfunction
  virtual function void build_map(); endfunction
endclass
```

Listing 1 illustrates both problems. In Listing 1(a), drawn from OpenTitan's verification library [1], a name-based rule searching for unsafe static OOP downcasts (Derived'(base_h), which bypasses SystemVerilog's runtime $cast check) matches the uvm_ prefix on uvm_reg_data_t'(csr_val) and emits 39 false-positive ERRORs. Symbol lookup resolves uvm_reg_data_t to a packed bit [63:0] typedef rather than a class. Conversely, in Listing 1(b), a base constructor calls non-virtual helper init_cfg(), which dispatches to virtual method build_map() before subclass fields are initialized (SV-OOP-017), while hard constraint c_lim leaves an empty 4-bit range for burst_len (SV-RND-030). Catching these defects statically without stalling in an SMT solver requires class-lineage call-graph summaries and polynomial-time interval deduction.

This paper presents **svlang** and **svlang-lint** (release 0.2.0), implemented in safe Rust. Throughout this paper, svlang-lint refers to the semantic static analyzer built on the svlang compiler SDK (workspace crate dv-lint, published as a CLI binary, Rust crate, and PyPI package svlang-lint), distinct from Hatta's syntax-tree linter dalance/svlint [8]. Because one workspace retains whitespace trivia, macro expansion coordinates, elaborated types and hierarchies, and word-level RTL/SVA netlists, compiler diagnostics, lint rules, source autofixes, formal back ends, and IDE services all work from the same front end.

**Contributions.** The main contributions of this work are:

1. **Pure-Rust SystemVerilog Front End and RTL/SVA Lowerer (§2):** svlang (76,086 LOC across 13 crates; zero external Cargo crates in the 12 Rust compiler, lowering, CLI, and WebAssembly crates, plus pyo3 in svlang-py) implements a ∼110 MB/s lossless lexer, Clause 22 preprocessor with dual token origins, arena CSTs, arbitrary-width four-state constant evaluation, IEEE 1800 §§11.6/11.8 expression typing, and word-level RTL plus Annex F SVA lowering to Btor2 [15], AIGER, and Yosys [11].

2. **Ten-Gate Differential Oracle Verification (§3):** Ten automated gates compare each compiler and linter stage against C++ Slang v12.1 [12] across an opt-in C ABI boundary, locking token streams, CST round-trips, 99,712 bound expression types, 172 UVM 2020.3.1 class specializations, lowered netlists, and diagnostic multisets alongside 12 simulator-dialect probes and a 90-kind AST coverage floor.

3. **Semantic Verification Linting and CST Repair (§4):** svlang-lint executes 433 rule implementations (461 catalog entries across 20 categories, including 52 Rust-native semantic rules) in parallel with per-Session isolation. It pairs polynomial-time constraint interval and difference-bound cycle deduction with class-lineage UVM objection and TLM-to-covergroup checks, RTL/CDC/FSM and definite-assignment dataflow, 38 CST autofix modules, Verissimo-compatible block waivers, and dependency-hashed run caching.

4. **Empirical Evaluation on Compliance, Open RTL, Defect Pairs, and Production DV (§5):** The evaluation covers CHIPS Alliance sv-tests [16] (1,024/1,024 pass), standalone lowering across 16 open-source RTL repositories (2,184 modules), 415 paired (bad.sv, good.sv) defect fixtures compared against Verible, Verilator, and Icarus Verilog (which together miss 70.6% of rules), and production UVM codebases from OpenTitan, Accellera uvm-core, and OpenHW core-v-verif.

## 2 Architecture of the svlang Compiler Front End

Figure 1 shows the crate dependency graph of the svlang workspace, and Table 1 lists the Rust line counts across all 20 crates (143,840 lines in total: 76,086 lines in the 13 compiler front-end, formal lowering, and SDK crates, and 67,754 lines in the 7 verification modeling, linting, LSP, CLI, and parity-oracle crates).

Two architectural rules govern the workspace. First, crates form a strict directed acyclic graph (DAG) with zero external Cargo dependencies across the 12 Rust compiler, lowering, CLI, and WebAssembly crates (sv-source through svlang-cli and svlang-wasm, 74,432 LOC). Lexing, preprocessing, parsing, arbitrary-width four-state arithmetic, elab-

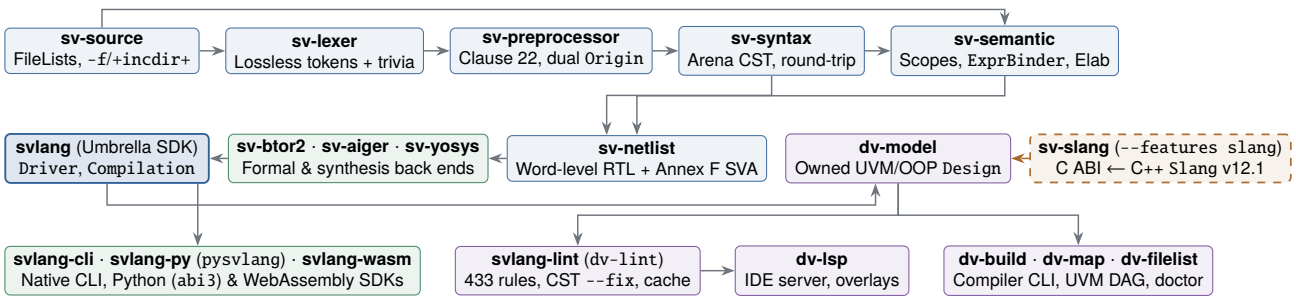


Figure 1: Crate dependency DAG of the `svlang` and `svlang-lint` Rust workspace. Blue and green nodes form the IEEE 1800-2023 compiler, formal lowering, and SDK pipeline (zero C++ or UVM coupling); purple nodes implement the UVM-aware static analyzer, language server, and command-line utilities; the dashed amber node (`sv-slang`) is an opt-in C ABI bridge to C++ `Slang` v12.1 compiled only for differential parity gates.

Table 1: Crate structure and Rust lines of code (LOC) across the 20-crate `svlang` and `svlang-lint` `0.2.0` workspace (commit `ba86329`).

| Crate | LOC | Role in Workspace |
|---|---|---|
| *Compiler Front End, Lowering & SDK (13 crates)* | | |
| `sv-source` | 1,088 | VCS/Xcelium filelists, env vars, simulator flags |
| `sv-lexer` | 1,042 | Lossless byte-exact lexer (~110 MB/s) |
| `sv-preprocessor` | 1,654 | Clause 22 macros, dual `Origin`, event table |
| `sv-syntax` | 6,109 | Event-built arena CST, byte round-trip |
| `sv-semantic` | 34,063 | Scopes, 4-state `consteval`, `ExprBinder`, elab |
| `sv-netlist` | 22,061 | Word-level RTL IR, Annex F SVA NFA/DFA |
| `sv-btor2` | 2,060 | Fail-closed Btor2 formal transition writer |
| `sv-aiger` | 1,859 | Bit-blasted AIGER 1.9 safety/justice writer |
| `sv-yosys` | 993 | Yosys JSON word-level synthesis bridge |
| `svlang` | 2,384 | Public Rust SDK (`Driver`, `Compilation`) |
| `svlang-cli` | 796 | Standalone `svlang` CLI (`check`, `stages`) |
| `svlang-py` | 1,654 | Python `abi3` bindings (`pysvlang` via `pyo3`) |
| `svlang-wasm` | 323 | Browser/WebAssembly target (0 external deps) |
| **Subtotal (Core)** | **76,086** | **0 external deps in 12 Rust crates + pyo3** |
| *Verification Model, `svlang-lint`, LSP & Parity Tools (7 crates)* | | |
| `dv-model` | 1,101 | Owned UVM/OOP class, phase & body model |
| `sv-slang` | 414 | Opt-in C ABI bridge to C++ `Slang` oracle |
| `dv-lint` (`svlang-lint`) | 61,963 | 433 rules, semantic caches, `--fix`, SARIF |
| `dv-lsp` | 3,554 | Stdio LSP server, CST index & live overlays |
| `dv-build` | 292 | Native compiler error, symbol & netlist CLI |
| `dv-map` | 226 | Static UVM component hierarchy extractor |
| `dv-filelist` | 204 | Multi-simulator filelist diagnostic checker |
| **Subtotal (DV)** | **67,754** | **Parallel static analysis & IDE stack** |
| **Workspace Total** | **143,840** | **20 Rust crates (Apache-2.0)** |

oration, word-level netlist lowering, Btor2/AIGER/Yosys emission, JSON/SARIF reporting, and LSP JSON-RPC framing are written directly against the Rust standard library (workspace MSRV 1.82, `edition = "2021"`; MSRV 1.85 in `svlang-py`). Across all 20 crates, the only third-party Cargo dependencies are `pyo3` in `svlang-py` for CPython `abi3` wheels and an ASCII-mode `regex` crate in `dv-lint` for user-supplied waiver and naming patterns. Second, none of the compiler or lowering crates (`sv-source` through `svlang`) contain UVM-specific rules or C++ code; C++ `Slang` is compiled only behind the optional `sv-slang` crate when `--features slang` is requested for parity gates.

### 2.1 Lossless Lexing, Dual-Origin Provenance, and Arena CST

A static analyzer that also rewrites source files (`--fix`) and serves live editor buffers faces two opposing requirements. Semantic passes need a clean stream of macro-expanded tokens, whereas source edits need every space, comment, and compiler directive preserved at its exact byte offset. Three front-end crates handle this split:

**Filelist Ingestion (`sv-source`).** Hardware verification builds rarely pass a plain list of files. Regression scripts mix nested `-f` and `-F` command files (where `-F` resolves paths relative to the enclosing filelist), `$VAR/${VAR}/$(VAR)` environment variables, include and library directories (`+incdir+`, `+define+`, `+undefine+`, `-y`, `-v`, `+libext+`, `-top`, `-ntb_opts uvm-x.y`), and simulator runtime options (`-l`, `+acc`). Rather than rejecting simulator-specific flags, `sv-source` normalizes them into an owned `FileList` with source spans and structured diagnostics.

**Lossless Tokenization (`sv-lexer`).** Every byte of an input buffer belongs to one token, including whitespace, comments, compiler directives, and macro operators. Token boundaries follow `Slang` (for example, keeping `[*` and `[->` as two-token sequences and separating integer base specifiers from digit runs). Across `uvm-core` plus repository test suites (952 files, 3.2 MB) and `sv-tests` (1,029 files), `sv-lexer` runs at roughly 110 MB/s and reproduces every input file byte-for-byte.

**Dual-Origin Macro Provenance and Preprocessor Event Table (`sv-preprocessor`).** `sv-preprocessor` implements IEEE 1800-2023 Clause 22, including parameterized `` `define `` defaults, token pasting (``` `` ```), stringification (`` `" ``), escaped quotes, `` `undefineall ``, `` `begin_keywords ``, and the 2023 compound boolean form of `` `ifdef/`elsif ``. Directives are stripped before parsing, while every emitted token carries two coordinates:

$$\text{Provenance}(t) = \big(\text{origin}(t),\ \text{written}(t),\ \text{is_macro}(t)\big) \quad (1)$$

Here $\text{origin}(t) = (f_{\text{exp}}, b_{\text{exp}})$ records the file index and byte offset of the outermost macro call in user code, while $\text{written}(t) =$

($f_{\text{def}}$, $b_{\text{def}}$) points to the exact token site inside a `'define` body or macro argument. In parallel, `sv-preprocessor` records a *Preprocessor Event Table* (`MacroDef` parameter/body spans, `IncludeEdge` edges, and `CondEvent` conditional regions) on `sv_syntax::Tree`, so macro-hygiene and include-guard rules do not need to re-scan raw text. `sv_semantic::Locator` indexes newline offsets once per file for $O(\log L)$ byte-to-(line, col) lookup. Declarations expanded inside UVM macros inspect their written tokens via written($t$), lint findings report at origin($t$) (column 0 of the macro invocation line), and automated CST fixes skip any edit touching a token marked is_macro($t$).

**Event-Constructed Arena CST and Compilation Units (`sv-syntax`).** `sv-syntax` uses recursive descent with precedence climbing and error recovery across the full IEEE 1800-2023 grammar, including Table 11-2 expression precedence, classes, constraints, covergroups, SVA sequences and properties, clocking blocks, checkers, UDPs, configs, `nettype`, and `randsequence`. Parser events populate a flat `Tree` backed by a contiguous token slice and an arena of `SyntaxKind`-tagged nodes, so completed subtrees can be wrapped in place without per-node heap allocations. Leading trivia stays attached to file tokens so `Tree::file_text` reconstructs the input text identically. To match different simulator build flows, `sv-syntax` offers three compilation-unit modes: default `parse_files` (one `Tree` per primary file with macro state carried forward across `-f` entries), `--single-unit` (`parse_unit`, merging all files into one compilation unit), and `--separate-units` (`parse_files_isolated`, enforcing IEEE 1800-2023 §3.12.1 per-file macro isolation).

### 2.2 Scopes, Four-State Constant Evaluation, and Context-Determined Typing

Under IEEE 1800-2023 §§11.6–11.8, expression bit widths and signedness cannot be resolved purely bottom-up: widths and signedness propagate both upward from operands and downward from the enclosing assignment target, port connection, concatenation, or ternary expression. `sv-semantic` (34,063 LOC) divides semantic analysis into four stages:

1. **Scope and Inheritance Graph (`scopes`, `names`, `iclass`):** A collection pass indexes declarations across `$unit`, packages (with chained `import` and `export` clauses), modules, interfaces, programs, classes, and subroutines. It binds out-of-block `extern` method and constraint bodies and checks IEEE 1800 §§8.25–8.26 parameterized class and `interface class` diamond-inheritance rules.

2. **Arbitrary-Width Four-State `consteval`:** Parameters, `localparam`s, `specparam`s, `min:typ:max` triplets, enum values, and unpacked dimensions are folded over arbitrary-width four-state bitvectors (`0`, `1`, `x`, `z`) backed by inline words and heap `bigint` storage. For IEEE 1800 §13.4.3 constant functions (such as custom `clog2` or lookup-table generators), `consteval` runs a tree-walking interpreter supporting local variables, branches, `for`/`while`/`repeat` loops, concatenations, replications, and `$clog2`/`$bits`. Non-constant inputs evaluate to *unknown* rather than silently substituting zero.

3. **Two-Pass `ExprBinder` and `TypeOracle`:** `ExprBinder` propagates self-determined and context-determined widths and signedness across unsized fill literals (`'0`, `'1`, `'x`, `'z`), unpacked array and queue concatenations, streaming concatenations (`{<<}`/`{>>}`), `type(expr)` and `type(this)`, `union soft` types, `nettype` over enums, inline `randomize(args) with` scopes, 2023 triple-quoted strings, and ANSI port-list `DataType` carry-forward. `TypeOracle` and `SpecModel` expose per-instance signal scopes and parameterized class specializations to downstream checks.

4. **Parallel Semantic Checks (`SemaChecks`):** Ten check groups (`names`, `checks`, `bind`, `iclass`, `classes`, `order`, `calls`, `typed`, `sva`, `udp`) execute across worker threads and merge into a deterministic `semantic_diags` vector.

### 2.3 Hierarchical RTL Elaboration and Word-Level Lowering

Starting from explicit or inferred top modules, `elaborate_full` constructs an `ElaboratedDesign`. It expands instance arrays, unrolls `generate` loops, evaluates `if`/`case` generate blocks with LRM `genblk<n>` naming, iterates `genvar`-dependent `bind` directives and `defparam` overrides to a fixpoint, and elaborates Clause 17 `checker`s, Clauses 28–31 UDPs (including `primitive p(.*)` wildcard ports), pull and array gates, Clause 33 `config` blocks, and Clause 6.6.7/6.6.8 user `nettype`s and `interconnect` nets.

`sv-netlist` (22,061 LOC) lowers `ElaboratedDesign` into a format-independent, word-level Static Single Assignment (SSA) RTL netlist:

- **Procedural Lowering and Driver Checks:** `always_ff` blocks (with asynchronous reset polarity extraction), `always_comb` blocks (with complete-assignment latch checks), `always_latch`, continuous `assign` statements, static loops, and function calls lower into word-level multiplexers, arithmetic/bitwise nodes, and registers (including `$anyseq`/`$anyconst` formal inputs). Bit-range interval tracking reports illegal multi-driver overlaps across procedural and continuous drivers, while Tarjan's strongly connected components (SCC) pass flags combinational loops.

- **Annex F SVA Automaton Compilation (`sva.rs`, `sva_nfa.rs`):** Bounded temporal properties without variable-length sequence gaps lower to a compact `$past` shift-register matcher. Unbounded sequences and full IEEE 1800-2023 Clause 16 / Annex F temporal operators cover goto repetition

([->m:n]), non-consecutive repetition ([=m:n]), unbounded delays (##[m:$], [*m:$]), strong(seq), bounded and unbounded property negation (not), ranged s_eventually [m:$], followed-by (#-#, #=#), and synchronous (sync_accept_on/sync_reject_on) plus asynchronous (accept_on/reject_on) aborts. These properties are normalized to Annex F negation normal form (NNF) and compiled into Glushkov/Thompson bit-parallel NFAs, dead-end-pruned Shannon ROBDD DFAs (ConsequentDfa, CoverDfa), antecedent-rejection (ant_reject) monitors, and Miyano–Hayashi breakpoint liveness trackers.

- **Back-End Writers:** From sv-netlist, sv-btor2 emits word-level Btor2 [15] transition systems (failing closed on unmodeled multi-clock or concurrent-assumption constructs rather than dropping constraints), sv-aiger bit-blasts safety and justice properties to AIGER 1.9, and sv-yosys writes Yosys [11] JSON netlists. Downstream, sv-btor2 also serves as the SystemVerilog front end for the InFormal model checker [17].

# 3 Differential Reference-Oracle Verification Methodology

The IEEE 1800-2023 standard spans more than 1,300 pages of interacting lexical, scoping, bit-sizing, and OOP specialization rules [4]. A clean-room front end tested only on small handwritten unit tests will inevitably diverge from established compilers on corner cases such as whitespace inside stringified nested macros, signedness propagation across ternary branches, or canonical type naming in mutually recursive UVM classes. To catch semantic mismatches early while keeping default builds free of C++, svlang uses a *ten-gate differential oracle pipeline* against C++ Slang v12.1.0 [12].

## 3.1 Opt-In C ABI Reference Isolation (sv-slang)

Rather than binding C++ headers through generated wrappers, sv-slang isolates Slang behind a compact versioned C ABI (CApi.h, ABI version 7). Crate dv-model defines an owned Rust Design structure holding packages, classes, type parameters, UVM roles (UvmKind), UVM phases (PhaseKind), fields, methods, resolved call and assignment facts, and macro expansion locations. dv_model::Design can be populated either natively from sv-semantic (default Cargo build, zero C++) or across sv-slang (--features slang). Performance gate R0 (dv-map --bench) confirms that FFI marshaling adds less than 10% wall-clock overhead on top of C++ AST construction.

## 3.2 Ten-Layer Differential Gate Pipeline

Table 2 summarizes the ten verification gates in tools/gates/, run alongside rust_gate.sh (cargo fmt --check and clippy -D warnings on both default and --features slang builds):

- **Lexical and Preprocessor Parity (pp_parity):** Verifies that sv-preprocessor emits the exact 278,335-token stream on Accellera uvm_pkg.sv and matches all 949 legal sv-tests files token-for-token.
- **CST Round-Trip and Precedence Parity (cst_parity):** Checks Tree::file_text byte-exact source reconstruction and prefix S-expression dumps (+ a (* b c)) across all 8,972 expressions in uvm-core, 849 clean sv-tests files, and 772 repository test files, while rejecting the same 128 sv-tests files rejected by Slang's parser.
- **Expression-Type Parity (expr_gate):** Compares the bound IEEE 1800 §§11.6/11.8 type string of every elaborated expression across 904 multi-file compilations (99,712 expressions). Driving the last 54 mismatches down to **0** (matched = 99,712, wrong = 0) exposed subtle bugs in per-instance sig_scope lookup, four-state propagation through type(expr), package export visibility, unpacked array localparam bounds, ANSI port data-type carry-forward, and queue concatenation typing.
- **Model, Symbol, Specialization, Netlist, and Lint Parity:** model_parity, sym_gate, spec_gate, netlist_gate, and lint_parity lock symbol tables, four-state constant values, class specializations, lowered netlists, and diagnostic multisets across commits.

## 3.3 Vendor-Dialect Probing and AST Coverage Floors

Production RTL and verification codebases frequently rely on simulator extensions that strict IEEE 1800-2023 forbids. Instead of copying claims from vendor manuals, vendor_table.py compiles 12 single-construct probe files (testdata/vendor/) through each installed tool (dv-build, svlang-lint, Slang, Verilator, Icarus Verilog, and commercial simulators where licensed) to record empirical acceptance tables and configure --compat-profile.

Parsing a construct does not guarantee that any lint rule inspects it. ast_coverage.py checks which AST node kinds are referenced by rules and enforces a monotonic floor (ast_coverage_floor.json) across 90 construct kinds: 23 SymbolKinds, 26 StatementKinds, 29 ExpressionKinds, 3 TimingControlKinds, 2 AssertionExprKinds, and 7 ConstraintKinds. Dropping below any category floor fails the test gate. In parallel, slang_canary.py tests upstream Slang commits so temporary compatibility shims are removed once upstream fixes land. Across UVM library releases (IEEE 1800.2-2020 and Accellera UVM 1.1d, 1.2, and 2020.3.1), API differences are read from embedded ApiPack tables generated from @uvm-ieee, @uvm-accellera, and @uvm-contrib source tags rather than hardcoded version checks.

# 4 Semantic Static Analysis and Automated Repair in svlang-lint

Figure 2 shows the execution pipeline of svlang-lint (crates/dv-lint, 61,963 LOC). Its 433 rule implementations correspond to 461 catalog entries (99 ERROR, 299 WARNING,

Table 2: Automated differential parity and correctness gates checking pure-Rust `svlang` and `svlang-lint` against the C++ `Slang` v12.1 reference front end.

| Gate Script | Layer Tested | Comparison Granularity | Verified Corpus Scale & Parity Result |
|---|---|---|---|
| `pp_parity.py` | `sv-preprocessor` | Exact token kind + text stream | `uvm_pkg.sv` (278,335 tokens) + 949 legal `sv-tests`: 100% token-identical |
| `cst_parity.py` | `sv-syntax` | Outline + prefix `(op a b)` AST + byte round-trip | `uvm-core` (168 files, 5,402 items, 8,972 exprs), 849 clean + 128 rejected `sv-tests`, 772 repo files |
| `model_parity.py` | `dv-model` | Canonical `Design::dump` (`--bodies`, `--types`) | 783 repo compiles (32,731 classes) class-identical; 353/353 `uvm_pkg` classes matched |
| `sym_gate.py` | `sv-semantic` | `--symbols`, `--specs`, `--types`, `--values`, `--hier` | 949/949 legal `sv-tests` and 783/783 repo compiles spec-, value-, and hierarchy-matched |
| `ieee_gate.py` | `dv-build` | Per-clause legal accept / `should_fail` reject | 948/949 legal + 75/75 negative `sv-tests` (1 preprocess-only, 0 gaps) |
| `slang_unit_gate.py` | `sv-semantic` | Diagnostic parity on `Slang`'s own unit tests | 0 false-positive errors across 20/21 upstream `Slang` AST unit-test areas (`false = 1`) |
| `expr_gate.py` | `ExprBinder` / `TypeOracle` | Every expression's bound IEEE 1800 §11.6 type | **99,712 / 99,712 expressions matched** (`wrong = 0`) across 904 compilations |
| `spec_gate.py` | `SpecModel` | Elaborated parameterized specialization counts | **172 / 172 identical** parameterized class specializations on UVM 2020.3.1 (`svlang_specdump`) |
| `netlist_gate.py` | `sv-netlist` | Word-level IR, memory blast & Annex F SVA DFAs | **55 / 55** RTL/fail-closed/CLI checks + 46/46 SVA property tests & 160-iter Annex F fuzzer |
| `lint_parity.py` | `svlang-lint` (`dv-lint`) | Multiset of `(file, line, col, rule, msg)` | 783/783 repo compiles identical across 409 shared rules; 0 UVM 2020.3.1 error-budget regressions |

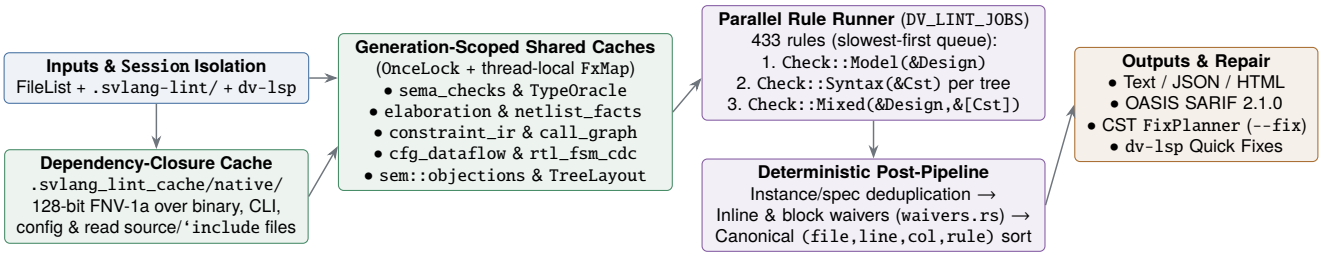


Figure 2: Execution, caching, and post-processing pipeline of `svlang-lint` (`dv-lint`) and `dv-lsp`. Worker threads pull slowest rules first and share generation-scoped semantic and `TreeLayout` analyses through thread-local `FxMap` lookups backed by `OnceLock` initialization. Findings are deduplicated across specializations, filtered through session-scoped waivers, and sorted canonically so output does not vary with thread count.

and 63 `INFO`) across 20 categories once configurable style, naming, and compiler sub-rules are counted (409 dual-engine parity-gated rules and 52 Rust-native semantic rules). Each rule declares the input view it needs via `Check`:

- `Check::Model(fn(&Design, &mut Sink))`: inspects the OOP/UVM `Design` model once per compilation;
- `Check::Syntax(fn(&Cst, &mut Sink))`: inspects an indexed per-file `Cst` (`sv-syntax` tree with parent links, per-`SyntaxKind` node lists, and a `Locator`);
- `Check::Mixed(fn(&Design, &[Cst], &mut Sink))`: inspects the cross-file `Design` model alongside all syntax trees.

`lint_all` schedules enabled rules across `DV_LINT_JOBS` worker threads (defaulting to available CPU cores), starting with the slowest rules first. Heavy intermediate passes shared across rules are cached in `shared.rs` by run generation $g$. These include `SemaChecks`, full RTL `elaboration`, `netlist_facts`, `constraint_ir`, `cfg_dataflow`, `rtl_fsm_cdc`, and `TreeLayout` (per-line visual columns, tabs, comments, multiline `‘define` spans, and preprocessor events in `layout.rs`). The first worker requesting $(\text{name}, k_0, k_1, g)$ initializes an `Arc<OnceLock<Value>>`, and subsequent lookups on the same thread hit a lock-free thread-local `FxMap`.

Because elaborated checks may visit a parameterized module or class multiple times, findings are deduplicated by (rule, position, message), filtered through waivers, and sorted by (file, line, column, rule). `dv_lint::Session` stores rule options, `--history` state, and `WaiverState` in thread-local session storage with zero mutable globals. The CLI handles `--guideline`, `--conformance`, `--dump-config`, `--library`, `--exclude`, `--compat-profile`, `--separate-units`, and simulator `-uvm`/`-uvmhome`/`-ntb_opts uvm` switches directly in Rust.

To keep initial runs focused on functional defects rather than formatting noise, the default preset enables four bug-finding profiles (`soundness`, `methodology`, `performance`, and `security`) while leaving `style`, `smells`, and `advisory` opt-in. The subsections below detail the four core analysis domains and the repair engine.

### 4.1 Constrained-Random Static Analysis (`constraint_ir`)

In constrained-random verification (IEEE 1800 Clause 18), contradictory constraints cause runtime `randomize()` failures, dead `soft` weights, or base-class bounds that derived tests cannot widen. Invoking a general bit-vector SMT solver during linting risks multi-second stalls on non-linear arithmetic or array constraints, hurting editor responsiveness. Instead, `svlang-lint` applies *sound, incomplete polynomial-time domain and graph deduction* across 35 randomization rules. It projects out the tractable subset of hard constraints whose unsatisfiability proves that the full constraint block has no legal solution, avoiding both solver timeouts and false alarms on unmodeled expressions.

**Single-Variable Interval Intersection (`SV-RND-030..032`, `SV-RND-040`).** For each `rand` scalar field $x$ of declared type $T$ with domain $\mathcal{D}(T) = [\ell_T, u_T]$ (such as $[0, 15]$ for `bit [3:0]` or $[-128, 127]$ for `byte`), `svlang-lint` folds constant parameter bounds via `const_int` and intersects unconditional hard atoms (relational comparisons, `inside {[a:b], c}`, negations `!`, and conjunctions `&&`) into a finite interval union $I_{\text{hard}}(x) \subseteq \mathcal{D}(T)$:

- **`SV-RND-030` (`ContradictoryConstraints`, `ERROR`):** Fires when $I_{\text{hard}}(x) = \emptyset$. Separate constraint blocks on a class are intersected together only when no code in the build calls `constraint_mode()` to disable blocks dynamically.
- **`SV-RND-031` & `SV-RND-032` (`SoftConstraint NeverHolds`, `DeadDistItem`, `WARNING`):** Fire when a `soft` constraint interval $I_{\text{soft}}(x)$ satisfies $I_{\text{hard}}(x) \cap I_{\text{soft}}(x) = \emptyset$, or when a `dist` weight item lies entirely outside $I_{\text{hard}}(x)$ and can never be selected.
- **`SV-RND-040` (`HardcodedBounds`, `WARNING`, advisory):** Flags non-`soft` literal bounds on `rand` fields in `virtual` or extended base classes, which prevent derived tests from widening stimulus ranges without disabling the whole base constraint block (`SV-LSP-001`).

**Multi-Variable Difference-Bound Cycles and Array Cardinality (`SV-RND-041..046`).** Module `constraint_ir.rs` pairs single-variable intervals with cross-variable structural checks cached per syntax tree:

1. **Relational Contradiction Cycles (`SV-RND-041`):** Unconditional hard inequalities ($u < v$, $u \le v$, $u > v$, $u \ge v$, $u == v$) form a directed weighted graph $G = (V, E)$ where edge $(u \xrightarrow{w} v)$ encodes $u + w \le v$, with $w = 1$ for strict inequalities ($<$) and $w = 0$ for non-strict inequalities ($\le$, $==$). Any directed cycle $C$ in $G$ with $\sum_{e \in C} w(e) \ge 1$ entails $u + 1 \le u$, proving unsatisfiability regardless of operand width.
2. **Unconstrained `foreach` Loops (`SV-RND-042`):** Using `TypeOracle`, `svlang-lint` identifies `rand` dynamic arrays and queues iterated by `foreach (arr[i])` when neither the class nor its ancestors constrain `arr.size()`. Without a size constraint, simulators leave `arr` empty and skip the `foreach` body silently.
3. **Dangling `disable soft` (`SV-RND-043`) and `randc` Hazards (`SV-RND-045/046`):** Walks the inheritance chain (`class_chain`) on `disable soft x` to flag fields lacking any inherited `soft` constraint, and checks `solve ... before` and inline `randomize() with` restrictions on cyclic `randc` variables.
4. **`unique` Pigeonhole Overflow (`SV-RND-044`):** On `unique {arr}` constraints over fixed-size unpacked arrays (or dynamic arrays constrained to constant `size() == K`) of 2-state integral or enum elements, `svlang-lint` computes array length $|A|$ and domain size $|\mathcal{D}(T)|$ in `u128`, flagging cases where $|A| > |\mathcal{D}(T)|$ (for example, `rand bit [1:0] id[5]; constraint c { unique {id}; }`, where $5 > 4$).

### 4.2 Inter-Procedural UVM, OOP, and Register-Model Verification

UVM and testbench methodology account for the largest rule group in `svlang-lint`: 100 UVM/RAL rules, 20 OOP rules, 16 Architecture rules, and 16 IEEE 1800.2 Deprecated-API rules. Many of these defects span multiple methods, inheritance layers, or component instances.

**Class-Lineage Objection Balance (`sem::objections`).** In UVM phases and sequences, an unmatched `raise_objection()` hangs simulation until timeout, whereas an unmatched `drop_objection()` terminates the phase prematurely. Checking each task in isolation raises false alarms on standard UVM idioms: sequences that raise an objection in `pre_body()` or `pre_start()` and drop it in `post_body()` or `post_start()`, base/subclass callback splits, and helper tasks. `sem::objections::Objections` summarizes $(n_{\text{raise}}, n_{\text{drop}})$ across every routine (including out-of-block `extern` definitions) of every class in the build. Before `UVM-OBJ-001`, `UVM-OBJ-003`, or `UVM-PHS-006` flags an unbalanced routine $r$ in class $C$, `Objections::elsewhere` checks whether another routine $r' \neq r$ in $C$, an ancestor of $C$, or a subclass of $C$ carries the opposite net imbalance (excluding routines that are already self-balanced). Within each routine's CFG, `svlang-lint` tracks distinct objection handles separately, pairs symmetric conditional guards (`if (c) raise ... if (c) drop`) and `if (phase != null)` checks, and flags early `return` statements inside loops while an objection is held.

**Transitive Constructor Virtual Calls and Override Contracts (`call_graph`).** Calling a `virtual` method inside `new()` dispatches to the derived override before derived-class fields are constructed. While `SV-OOP-005` catches direct virtual calls in `new()`, `call_graph.rs` implements `SV-OOP-017`

(TransitiveVirtualCallInNew) via a bounded breadth-first search (call depth $\leq$ 8, class lineage $\leq$ 64) over dv_model::Design call summaries, starting from non-virtual helper calls on this (excluding super. calls) in new(). Alongside this check, SV-OOP-016 verifies IEEE 1800 §8.20 default-argument matching across virtual overrides, and SV-OOP-018 (MissingSuperInDoHook) flags do_copy, do_compare, do_print, do_pack, and do_unpack overrides that omit super.do_*() and silently drop base-class fields.

**TLM Connections, Factory, Callbacks, uvm_config_db, and RAL Maps.** Across the component hierarchy, svlang-lint checks methodology invariants that standard compilers treat as ordinary method calls or untyped strings:

- **Monitor-to-Covergroup Wiring (UVM-TLM-007):** In testbenches that declare functional covergroups, UVM-TLM-007 traces each transaction type $T$ published on a uvm_monitor's uvm_analysis_port#(T) and verifies that at least one class holding a non-empty covergroup (directly, by inheritance, or via a member handle) receives $T$ through uvm_subscriber#(T), an analysis imp/export, or a TLM analysis FIFO.
- **Factory, Callbacks, Sequencer, and uvm_config_db Checks:** UVM-FAC-006 verifies that type-based factory overrides (set_type_override_by_type, set_inst_override_by_type, type_id::set_type_override) supply an override class derived from the requested base type. UVM-CBK-001 checks that uvm_callbacks#(T,CB)::add calls have matching ‘uvm_register_cb(T,CB) registrations, UVM-SEQ-017 checks seq.start(seqr) sequencer types against p_sequencer, and UVM-PHS-012 flags copy-pasted super.<other_phase>(phase) calls. For configuration wiring, UVM-CFG-001, UVM-CFG-005, and UVM-DAG-001 match uvm_config_db#(T)::set/get types and instance glob paths against the component tree while exempting unspecialized type parameters such as CFG_T.
- **Register Abstraction Layer (RAL) Interval Checks:** RAL-FLD-001 computes [lsb_pos : lsb_pos + size - 1] bit intervals across uvm_reg_field::configure() calls inside uvm_reg::build() to catch overlapping bit slices. At the map level, RAL-MAP-002 and RAL-COL-001 check byte-address ranges for collisions and RAL-MAP-001 flags unmapped registers in uvm_reg_map.

### 4.3 RTL, Clock-Domain, Dataflow, and Virtual-Interface Checks

Building directly on sv-semantic and sv-netlist allows svlang-lint to run RTL design checks and UVM testbench checks in one pass:

- **Clocked Blocking Races, NBA Container Hazards, and CDC (SV-RCE-002, SV-CON-016/017, rtl_fsm_cdc):** Following Cummings' guidelines on Verilog scheduling races [18] (IEEE 1800 §4.7), SV-RCE-002 flags variables written with blocking assignment (=) in one clocked always block and read in another block triggered by the same clock edge. SV-CON-017 flags illegal non-blocking assignments (<=) to dynamic arrays, queues, associative arrays, or class handles. Module rtl_fsm_cdc.rs extracts register clock domains and combinational fan-in cones with TypeOracle bit widths to catch unsynchronized multi-bit buses and combinational logic feeding single-bit clock-domain crossings (SV-CDC-001 [19]), mixed active-high/active-low reset polarities on one reset net, incomplete always_ff reset assignments, combinational clock gating, and module port-width mismatches (SV-RTL-001..005).
- **FSM Reachability (SV-FSM-002, SV-FSM-003):** On enum-typed state registers in always_ff/always_comb finite-state machines, rtl_fsm_cdc builds the state-transition graph from the reset state and reports unreachable states (SV-FSM-002) and non-idle deadlock states with no outgoing transitions (SV-FSM-003).
- **Definite-Assignment Dataflow (cfg_dataflow):** Across functions and tasks, cfg_dataflow.rs runs forward path-sensitive analysis over if/else, case (tracking default arms), loops, and early returns. It flags non-void functions that can fall through without returning a value (SV-DAT-001), reads of definitely uninitialized locals (SV-DAT-002), and output arguments left unassigned on a return path (SV-DAT-003). Subroutines containing unexpanded statement macros other than standard ‘uvm_info/warning/error calls are skipped to avoid false alarms on hidden control flow.
- **Declaration-Resolved Virtual-Interface Drives:** Rules inspecting interface signal drives (SV-INT-001, SV-LANG-005, SV-DRV-001, UVM-VIF-DIR, and SV-OOP-008) resolve target declarations in sv-semantic instead of matching vif. or _if substrings. An assignment to cfg.bus.drv.data is recognized as an interface clocking-block drive when bus is declared virtual bus_if (including through typedefs or inherited members), whereas a struct variable named local_vif or an IEEE 1800 §8.26 interface class named printable_if is ignored.

### 4.4 Provenance-Guarded CST Autofixes, Waivers, Caching, and dv-lsp

**Provenance-Guarded CST FixPlanner (fix.rs).** Thirty-eight rule modules (43 catalog rule IDs) attach byte-span Fix edits (path, start..end, replacement text, and FixSafety). Behavior-preserving rewrites (FixSafety::Safe, applied by --fix) migrate deprecated IEEE 1800.2 starting_phase accesses to get_starting_phase() and set_starting_phase(...) (UVM-DEP-005), replace non-blocking <= writes to class handles or dynamic containers with = (SV-CON-017), rewrite $O(N)$ queue concate-

nation appends `q = {q, elem}` to `q.push_back(elem)` (`SV-PRF-007`), parenthesize macro parameters (`SV-STY-022`), split multi-declaration lines (`SV-STY-021`), and normalize block indentation (`SV-STY-020`). Behavioral defect repairs (`FixSafety::Unsafe`, gated behind `--unsafe-fixes`) convert blocking virtual-interface drives in run-time tasks to `<=` (`SV-INT-001`), replace `==`/`!=` against `'x`/`'z` literals with `===`/`!==` (`SV-LANG-001`), insert `virtual` on class interface handles (`SV-OOP-008`), and add missing `super.<phase>_phase(phase)` calls (`UVM-PHS-001`). Judgment-dependent hints such as covergroup bin thresholds (`SV-COV-011`) use `FixSafety::DisplayOnly`.

Before modifying any file, `FixPlanner` enforces three safety invariants. First, any edit touching a macro-expanded token (origin ≠ written or `is_macro`) is downgraded to display-only so `` `define `` bodies and call sites are never corrupted. Second, files with compiler errors (`SV-CMP-001`) are left untouched. Third, overlapping edits within one pass are deferred to the next re-parse and re-lint pass, bounded at 10 passes with edit-cycle detection and 3-line unified diffs on `--fix-dry-run`.

**Scoped Block Waivers (`waivers.rs`).** To support incremental rollout on legacy codebases, `svlang-lint` implements inline waivers (`svlang-lint:waive`, `svlang-lint-disable-next-line`, `lint:off`) and Verissimo-compatible block waivers (`svlang-lint:off/on`, `verissimo_lint:waive_start/waive_stop/waive_line`, and `lint:waive-start/stop`) with optional rule IDs and message regex filters. Waiver directives must appear immediately after `//` or `/*` (`comment_tail`), preventing ordinary prose comments that mention waiver syntax from suppressing findings or triggering unclosed-block errors (`SV-WVR-001`). Malformed waiver regexes report non-suppressible `UVM-WV-001` diagnostics.

**Dependency-Closure Result Cache and `dv-lsp`.** Enabled by default under `.svlang_lint_cache/native/`, the Rust result cache (`cache.rs`) computes a 128-bit two-lane FNV-1a key over the tool version, executable size and modification time, CLI flags, steering files (project configs, `-f`/`-F` filelists, waiver and baseline files, and Python plugins), and the canonicalized bytes of every source, transitive `` `include ``, and library file read during compilation. In the earlier C++ parity engine, splitting per-unit cache entries from 26 whole-program rules still required parsing and building the cross-file `Design` model on partial hits. `cache.rs` instead stores the pre-waiver finding vector (including autofix spans) as a single verified run entry and re-applies project config, waivers, and baselines directly on a cache hit. Updating waivers or baselines therefore skips re-compilation entirely, while any source modification falls back cleanly to a full run.

For CI code-scanning pipelines, `report.rs` emits OASIS SARIF 2.1.0 [20] reports with rule descriptions, remediation metadata, related spans, and fix payloads. For interactive IDE use, `dv-lsp` implements an LSP [6] server directly over `std::io` JSON-RPC without external protocol crates. It indexes workspace `.sv`/`.svh`/`.v`/`.vh` files in parallel (~50 ms across the 170 files of Accellera `uvm-core`) to serve go-to-definition, hover types, and completion while routing unsaved editor buffers through `sv_preprocessor::overlay` for live diagnostics and quick fixes.

Table 3: RQ1 conformance of `svlang` on CHIPS Alliance `sv-tests` (`415ba5f934ef`; 4 runtime-simulation-only tests excluded from 1,028).

| Clause | IEEE 1800 Topic | Run | Pass | FR / FA | Rate |
|---|---|---|---|---|---|
| 5 | Lexical conventions | 50 | 50 | 0 / 0 | 100% |
| 6 | Data types | 84 | 84 | 0 / 0 | 100% |
| 7 | Aggregate data types | 104 | 104 | 0 / 0 | 100% |
| 8 | Classes (OOP) | 53 | 53 | 0 / 0 | 100% |
| 9 | Processes | 46 | 46 | 0 / 0 | 100% |
| 10 | Assignment statements | 10 | 10 | 0 / 0 | 100% |
| 11 | Operators & expressions | 87 | 87 | 0 / 0 | 100% |
| 12 | Procedural statements | 27 | 27 | 0 / 0 | 100% |
| 13 | Tasks & functions | 15 | 15 | 0 / 0 | 100% |
| 14–15 | Clocking & sync | 10 | 10 | 0 / 0 | 100% |
| 16 | Assertions (SVA) | 49 | 49 | 0 / 0 | 100% |
| 18 | Constrained random | 134 | 134 | 0 / 0 | 100% |
| 20–21 | Utility & I/O system tasks | 76 | 76 | 0 / 0 | 100% |
| 22 | Compiler directives | 75 | 75 | 0 / 0 | 100% |
| 23–26 | Hierarchy, prog., ifc., pkg. | 7 | 7 | 0 / 0 | 100% |
| — | Generic, sanity, UVM TBs | 197 | 197 | 0 / 0 | 100% |
| **All** | **Total (0 crashes/timeouts)** | **1,024** | **1,024** | **0 / 0** | **100.0%** |

## 5 Experimental Evaluation

The empirical evaluation addresses four research questions:

- **RQ1 (Language Conformance & Expression-Typing Parity):** How completely does pure-Rust `svlang` conform to IEEE 1800-2023 across public compliance suites, clause probes, and context-determined expression typing?
- **RQ2 (RTL Elaboration Coverage & Compiler Speed):** What fraction of open-source RTL modules does `svlang` elaborate and lower to word-level netlists and Btor2, and how do its speed and memory footprint compare with C++ `Slang`?
- **RQ3 (Defect Discrimination vs. Open-Source Baselines):** Across paired buggy and fixed SystemVerilog/UVM fixtures, how many defect rules are separated by existing open-source linters and simulators compared with `svlang-lint`?
- **RQ4 (Precision, Autofix Safety & Lint Turnaround on Production DV):** Does semantic resolution eliminate false-positive `ERROR`s on production UVM codebases while delivering compile-clean CST autofixes and fast cached re-runs?

### 5.1 RQ1: IEEE 1800-2023 Conformance and Oracle Parity

Table 3 reports `svlang`'s clause-by-clause conformance on the CHIPS Alliance `sv-tests` corpus [16] (commit `415ba5f934ef`). Four of the 1,028 handwritten cases under `tests/` require runtime simulation and are

skipped. Across all **1,024** remaining positive and negative (`:should_fail_because:`) cases, `svlang check` (and `svlang preprocess` on preprocessor-only tests) passes **1,024/1,024 (100.0%)**, with **0** false rejections, **0** false acceptances, **0** crashes, and **0** timeouts.

Beyond `sv-tests`, all **23** IEEE 1800-2023 clause-group probe files (`testdata/conformance/lrm/`, spanning Clauses 5–35 plus 2023 triple-quoted strings and compound preprocessor conditionals; runtime SDF Clause 32 and encrypted Clause 34 are out of scope), all **4** positive/negative 2023 construct suites (`testdata/conformance/ieee1800_2023/`), and all **18/18** conformance and census regressions (`ieee_and_census_fixes.rs`) pass cleanly. On `ieee_gate.py` (`dv-build` vs. `Slang`), `svlang` accepts **948/949** legal `sv-tests` files with zero native false errors (both `Slang` and `dv-build` reject the single preprocess-only Clause 22 test as a design unit, whereas `svlang preprocess` accepts it) and rejects all **75/75** negative tests (`0` gaps).

On the differential typing and specialization gates, `svlang` matches `Slang` v12.1.0 across **99,712/99,712** bound expressions (`wrong = 0`) in 904 compilations (`expr_gate.py`), matches **172/172** elaborated parameterized class specializations on Accellera UVM 2020.3.1 (`spec_gate.py`), and reports zero false-positive errors across **20/21** upstream `Slang` AST unit-test areas (`slang_unit_gate.py`). On the 12 non-standard vendor-dialect probes (`vendor_table.py`), strict `Slang` rejects all 12 unless explicit `--allow-*` or `--relax-*` flags are passed (11 flag-gated probes plus `declarations-after-statements`, which `Slang` always rejects). `Icarus Verilog` 13.0 accepts 1 (`genblk-reference`, down from 2 in version 12.0 after tightening `use-before-declare`), `Verilator` 5.052 accepts 5 (`array-concat-pattern`, `bare-value-param`, `genblk-reference`, `relax-string`, and `use-before-declare`, compared with 6 in version 5.020), and native `dv-build` / `svlang-lint` accepts 7 common simulator relaxations while rejecting the remaining 5.

### 5.2 RQ2: Open-Source RTL Elaboration Coverage and Compiler Throughput

Table 4 summarizes the open-RTL census (`census.py`) and stage-timing benchmark (`lowering_bench.py`, median of 5 single-process runs per top on an AMD Ryzen 7 8745HS) across **16 open-source RTL repositories**. The benchmark suite spans RISC-V processors (`picorv32`, `serv`, `biriscv`, `scr1`, `ibex`, `cv32e40p`, `cv32e40x`, `cva6`), interconnects and reusable building blocks (`verilog-axi`, PULP `axi`, `common_cells`, `tech_cells_generic`, `cvfpu`, `basejump_stl`), and silicon roots of trust (`caliptra-rtl` [2], `opentitan` [1]).

Every one of the **2,184** modules is elaborated on its own as a top-level design using default parameter values. Both `svlang` and `Slang` v12.1 reject 611 modules that cannot stand alone without parameter overrides or vendor macro headers (such as parameterized `basejump_stl` templates that intentionally omit defaults), leaving **1,573** measured standalone tops:

- **Elaboration and Lowering Coverage:** Under strict default flags, the 1,573 measured tops partition into **1,081 (68.7%)** valid lowered tops, 360 blocked by unsupported lowering constructs, 102 stricter or false rejections, 5 memory-outs (> 4.0 GB on wide priority encoders), and 25 lowered by `svlang` where strict `Slang` rejects the input (**0 crashes**, **0 timeouts**: 9 unknown vendor directives/macros, 7 default-parameter `$error` tasks, and 9 missing undeclared/bounds/member diagnostics). Among the 462 unlowered valid tops (360 + 102), 3 stop at parsing, 9 at elaboration, and 450 in `sv-netlist` lowering. Waiving OpenTitan's simulation-only `` `ASSERT_INIT_NET `` assertions (215 tops) and four-state `$isunknown` guards (108 tops) via `--skip-initial-checks --isunknown-is-false` raises lowering coverage to **1,359 / 1,571 (86.5%)**. Across all 1,106 lowered netlists (1,081 + 25), `sv-btor2` emits 976 by default and 78 more with declared clocks/properties (**1,054 / 1,106 = 95.3%**), refusing 52 multi-clock, array-init, or latch-loop modules.
- **Stage Timing and Memory:** Across the 1,081 lowered tops (1,571,434 compiled source lines counting shared headers per top; 1,630,984 SSA netlist values), all four stages finish in **32.18 s** (**48,834 lines/s**): 3.01 s in parsing (9%), 7.05 s in elaboration (22%), 21.73 s in word-level lowering (68%), and 0.39 s in Btor2 emission (1%). Median compile-and-lower time per top is **4.9 ms**, the largest register-block top (`caliptra-rtl soc_ifc_reg`, 11,217 lines) takes **2.27 s**, and peak RSS stays ≤ **350** MB. On front-end checking alone (`svlang check` vs. C++ `Slang` v12.1), `svlang` finishes all 1,081 tops in **12.32 s** versus `Slang`'s **8.86 s** (0.7× aggregate speed; faster or tied on 9/16 repositories, 1.1×–2.3× unrounded, where process startup and arena locality dominate).

### 5.3 RQ3: Defect Discrimination vs. Open-Source Linters and Simulators

To measure how much of `svlang-lint`'s bug catalog is already covered by existing open-source tools, `eval_baselines_on_rules.py` runs `Verible v0.0-4053` [7], `Verilator 5.052` [9], and `Icarus Verilog 13.0` [10] across all **415** paired `(bad.sv, good.sv)` rule fixtures in `testdata/rules/`. The remaining 46 of 461 catalog IDs are configurable sub-rules under `SV-STY-100`, `SV-NAM-100`, and `SV-CMP-001` tested in Rust unit suites.

**Evaluation Protocol.** Running standalone linters and RTL simulators on verification fixtures requires care so baselines are not penalized for testbench scaffolding. In `eval_baselines_on_rules.py`, `Verilator` receives `--lint-only --timing -Wall -I<testdata/common>` for UVM declarations, an empty `_bench_top` module so class-only files do not fail with *"No top level module"*, and flags suppressing fixture-scaffolding warnings (`-Wno-DECLFILENAME`, `-Wno-EOFNEWLINE`, `-Wno-UNUSED`, `-Wno-UNDRIVEN`, `-Wno-MULTITOP`, `-Wno-VARHIDDEN`). For `Verible`, trivial whitespace checks (`line-length`, `no-trailing-spaces`, `posix-eof`, `no-tabs`) are excluded so accidental line-length differences between `bad.sv` and

Table 4: RQ2 open-source RTL elaboration/lowering census and single-process compilation speed across 16 silicon repositories (AMD Ryzen 7 8745HS, release build; reference front end: C++ Slang v12.1.0).

| | Module Census (Strict Defaults) | | | | svlang Stage Timing & Throughput (Lowered Tops) | | | | | | Check Wall (s) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Repository** | **Mods** | **Meas.** | **Low.** | **Cov.** | **Lines** | **Parse** | **Elab** | **Lower** | **All (s)** | **Lines/s** | **svlang** | **Slang** |
| picorv32 | 21 | 15 | 11 | 73% | 25,687 | 0.04 | 0.03 | 0.07 | 0.15 | 176,916 | 0.10 | 0.08 |
| serv | 73 | 32 | 27 | 84% | 16,173 | 0.02 | 0.03 | 0.06 | 0.12 | 136,516 | 0.10 | 0.15 |
| biriscv | 46 | 42 | 34 | 81% | 15,439 | 0.03 | 0.03 | 0.10 | 0.17 | 92,765 | 0.11 | 0.17 |
| verilog-axi | 55 | 55 | 47 | 85% | 35,030 | 0.05 | 0.10 | 1.29 | 1.44 | 24,247 | 0.23 | 0.27 |
| scr1 | 38 | 37 | 29 | 78% | 11,341 | 0.03 | 0.04 | 0.07 | 0.14 | 82,723 | 0.11 | 0.15 |
| tech_cells_generic | 12 | 12 | 9 | 75% | 1,200 | 0.00 | 0.00 | 0.00 | 0.01 | 171,331 | 0.02 | 0.04 |
| common_cells | 200 | 194 | 182 | 94% | 86,719 | 0.19 | 0.23 | 1.19 | 1.63 | 53,140 | 0.70 | 1.06 |
| axi (PULP) | 118 | 7 | 6 | 86% | 17,274 | 0.05 | 0.02 | 0.03 | 0.10 | 179,876 | 0.08 | 0.07 |
| cvfpu | 12 | 9 | 7 | 78% | 19,392 | 0.03 | 0.04 | 0.10 | 0.18 | 107,159 | 0.09 | 0.09 |
| ibex | 30 | 27 | 10 | 37% | 14,819 | 0.02 | 0.11 | 0.14 | 0.27 | 54,545 | 0.16 | 0.07 |
| cv32e40p | 123 | 112 | 96 | 86% | 61,716 | 0.08 | 0.27 | 1.02 | 1.39 | 44,414 | 0.52 | 0.58 |
| cv32e40x | 42 | 32 | 31 | 97% | 56,656 | 0.06 | 0.38 | 0.47 | 0.91 | 61,981 | 0.52 | 0.22 |
| cva6 | 84 | 16 | 8 | 50% | 25,048 | 0.04 | 0.25 | 0.28 | 0.57 | 43,905 | 0.32 | 0.07 |
| caliptra-rtl [2] | 401 | 374 | 297 | 79% | 464,819 | 1.05 | 2.99 | 12.53 | 16.81 | 27,645 | 4.71 | 3.05 |
| opentitan [1] | 615 | 580 | 263 | 45% | 704,791 | 1.28 | 2.50 | 4.34 | 8.20 | 85,980 | 4.46 | 2.66 |
| basejump_stl | 314 | 29 | 24 | 83% | 15,330 | 0.04 | 0.02 | 0.03 | 0.09 | 161,827 | 0.10 | 0.13 |
| **Total / Aggregate** | **2,184** | **1,573** | **1,081** | **68.7%** | **1,571,434** | **3.01** | **7.05** | **21.73** | **32.18** | **48,834** | **12.32** | **8.86** |

good.sv do not count as bug detection; Table 5 lists both total substantive discrimination (lint rules plus syntax rejections) and lint-rule-only counts. Each baseline is credited with discriminating a fixture pair whenever it emits any substantive warning or error on bad.sv and none on good.sv.

**Results.** Table 5 gives the breakdown across all 20 categories:

- **Icarus Verilog 13.0** fails to compile good.sv on 375/415 pairs (90.4%) due to unsupported OOP, UVM, constraint, and SVA constructs, discriminating **2/415 (0.5%)** rules.
- **Verible v0.0-4053** parses good.sv without syntax errors on all 415 pairs (359/415 substantively clean), flags 113 bad.sv files, and discriminates **70/415 (16.9%)** rules (62 via lint rules and 8 via syntax rejection), mostly in Style (9/21), SVA syntax/labels (10/14), Deprecated syntax (8/15), and syntactic Randomization checks (13/32).
- **Verilator 5.052** compiles good.sv cleanly on 396/415 pairs (19 good.sv compile errors, reflecting Verilator 5's --timing and class support), flags 85 bad.sv files, and discriminates **71/415 (17.1%)** rules, concentrated in RTL/width Correctness (14/42), basic UVM/class checks (14/98), OOP (6/20), Coverage (6/18), and Types (4/6).
- **Combined Union:** Verible and Verilator overlap on 20 rules (Verilator and Icarus overlap on 1; Verible and Icarus on 0). Running all three baselines together in CI discriminates **122/415 (29.4%)** defect classes and misses **293/415 (70.6%)**, including **83.7%** (82/98) of UVM/RAL defects, **93.8%** (15/16) of Architecture defects, **72.4%** (21/29) of Concurrency/race defects, **66.7%** (12/18) of Coverage defects, **65.0%** (13/20) of OOP defects, and **64.3%** (27/42) of Correctness/CDC/FSM/dataflow defects.

### 5.4 RQ4: Precision, Autofix Safety, and Cached Lint Turnaround

Catching synthetic bugs is of little practical use if an analyzer floods verified UVM base libraries with false errors [14]. To test false-positive behavior under worst-case rule volume, every catalog rule and experimental check (--no-config --no-cache --preset all --experimental) was run across three open-source DV codebases:

1. **OpenTitan Shared DV Libraries [1]:** This suite covers dv_lib, dv_utils, csr_utils, dv_base_reg, push_pull_agent, common_ifs, mem_model, mem_bkdr_util, and str_utils (~7,900 non-blank lines vendored in ibex-dv and compiled through its filelist). Because IP testbenches across OpenTitan inherit from these base components, ERROR findings there represent analyzer false positives. Table 6 breaks down all nine root causes: replacing lexical heuristics with symbol, phase, and control-flow resolution reduced ERROR findings on OpenTitan's shared DV libraries from **170 to 0** (total findings under --preset all --experimental moved from 5,428 to 5,248, of which 3,015 are off-by-default INFO magic-number notes from SV-STY-010 on CSR bit widths). Across all of ibex-dv, ERROR findings fell by 64.7%, from **586 to 207**.
2. **Accellera uvm-core (IEEE 1800.2 Reference Library):** Single-routine objection checks initially emitted **8 false-positive ERRORs** in uvm-core's sequence and phase classes. The class-lineage sem::objections summary reduced uvm-core objection ERRORs from **8 to 0**, and all 52 native semantic rules introduced **0 error-budget regressions** on UVM 2020.3.1.
3. **Coverage-Gap Finding in OpenHW core-v-verif [3]:** Running TLM-to-coverage analysis (UVM-TLM-007, UncoveredAnalysisPort) on core-v-verif flagged agent monitor transaction types published on

Table 5: RQ3 defect discrimination across all 20 rule categories (461 catalog entries; 415 paired (`bad.sv, good.sv`) fixtures). A tool *discriminates* a rule iff it flags `bad.sv` and accepts `good.sv` without substantive warnings or errors.

| | Catalog | Paired | Rules Discriminated (bad flagged ∧ good clean) | | | | Baseline | Missed by |
|---|---|---|---|---|---|---|---|---|
| **Rule Category** | **Rules** | **Fixtures** | **svlang-lint** | **Verible (Lint)** | **Verilator** | **Icarus** | **Union** | **All Baselines** |
| UVM (Phases, Factory, TLM, RAL, Config) | 100 | 98 | 98 | 5 (5) | 14 | 0 | 16 (16.3%) | **82 (83.7%)** |
| Style (Layout, Macros, Labels) | 52 | 21 | 21 | 9 (9) | 3 | 0 | 11 (52.4%) | 10 (47.6%) |
| Correctness (RTL, CDC, FSM, Dataflow) | 44 | 42 | 42 | 4 (3) | 14 | 1 | 15 (35.7%) | **27 (64.3%)** |
| Randomization (Constraints, `dist`, `soft`) | 35 | 32 | 32 | 13 (13) | 4 | 0 | 16 (50.0%) | **16 (50.0%)** |
| Advisory | 33 | 31 | 31 | 0 (0) | 0 | 0 | 0 (0.0%) | **31 (100.0%)** |
| Maintainability | 32 | 32 | 32 | 1 (1) | 2 | 0 | 3 (9.4%) | **29 (90.6%)** |
| Concurrency (Races, Forks, VIF Drives) | 30 | 29 | 29 | 4 (3) | 5 | 0 | 8 (27.6%) | **21 (72.4%)** |
| OOP (Inheritance, `$cast`, Virtuals) | 20 | 20 | 20 | 3 (2) | 6 | 0 | 7 (35.0%) | **13 (65.0%)** |
| Coverage (Bins, Crosses, Sampling) | 20 | 18 | 18 | 1 (0) | 6 | 0 | 6 (33.3%) | **12 (66.7%)** |
| Performance (Queues, Boxing, Loops) | 17 | 17 | 17 | 3 (2) | 4 | 1 | 7 (41.2%) | 10 (58.8%) |
| Architecture (Layers, Packages, Coupling) | 16 | 16 | 16 | 0 (0) | 1 | 0 | 1 (6.2%) | **15 (93.8%)** |
| Deprecated (IEEE 1800.2 Migration) | 16 | 15 | 15 | 8 (8) | 3 | 0 | 9 (60.0%) | 6 (40.0%) |
| SVA (Vacuity, Clocks, Resets, Labels) | 14 | 14 | 14 | 10 (8) | 4 | 0 | 11 (78.6%) | 3 (21.4%) |
| DeadCode | 12 | 12 | 12 | 3 (3) | 0 | 0 | 3 (25.0%) | 9 (75.0%) |
| Types | 6 | 6 | 6 | 1 (0) | 4 | 0 | 4 (66.7%) | 2 (33.3%) |
| Portability | 6 | 5 | 5 | 2 (2) | 1 | 0 | 2 (40.0%) | 3 (60.0%) |
| Naming | 3 | 3 | 3 | 2 (2) | 0 | 0 | 2 (66.7%) | 1 (33.3%) |
| Security | 2 | 2 | 2 | 0 (0) | 0 | 0 | 0 (0.0%) | 2 (100.0%) |
| DPI | 2 | 1 | 1 | 0 (0) | 0 | 0 | 0 (0.0%) | 1 (100.0%) |
| RTL_TB_Bridge | 1 | 1 | 1 | 1 (1) | 0 | 0 | 1 (100.0%) | 0 (0.0%) |
| **All 20 Categories** | **461** | **415** | **415 (100%)** | **70 (62)** | **71 (17.1%)** | **2 (0.5%)** | **122 (29.4%)** | **293 (70.6%)** |

`uvm_analysis_port` whose connected subscriber classes compiled cleanly in simulation but still contained empty `TODO` covergroup stubs, leaving functional coverage uncollected.

**Lint Speed, Cache Replay, and CST Autofix Safety.** Table 7 summarizes lint turnaround and automated repair results (`eval_autofix_and_cache.py`, median of 5 runs). On Accellera `uvm-core` (170 files, 86,476 lines compiled through `uvm_pkg.sv`), `svlang-lint` expands Clause 22 macros, elaborates types and classes, and executes default bug-finding rules in **1.43 s** (**60,312 lines/s**), or all 433 rules (`--preset all`) in **1.48 s**, matching `Verilator 5.052 --lint-only --timing` (**1.41 s**) while running full UVM and constraint checks. On unchanged source closures (such as pre-commit hooks, waiver updates, or CI re-runs), `.svlang_lint_cache` replays `uvm-core` in **60.6 ms** (23.7× speedup), an 80-package, 2,000-class constrained-random suite in **8.1 ms** (24.9× speedup), and 80 standalone units in **9.6 ms** (14.5× speedup).

Across the 35 paired rule fixtures with autofix rules, `FixPlanner` rewrites 30 compile-clean `bad.sv` files (15 under safe-only `--fix` and 15 more with `--unsafe-fixes`). All **30/30 (100.0%)** modified files compile with zero `SV-CMP-001` errors, **29/30 (96.7%)** settle to a fixed point in one invocation, and **27/30 (90.0%)** clear all target-rule findings automatically. On `SV-RDB-005` and `SV-STY-010`, `FixPlanner` rewrites unambiguous expressions while leaving context-dependent literals (such as unnamed decimal widths) for manual review; on `UVM-PHS-001`, `dv-model` at `ba86329` records `super.build_phase(phase)` under a qualified call name rather than bare `build_phase`, so `FixPlanner`'s 10-pass cap halts repeated insertion cleanly with zero compile errors. On the remaining 5 fixtures, `SV-COV-011` emits a `DisplayOnly` bin-threshold hint and 4 standalone `bad.sv` files (`SV-COR-005`, `SV-OOP-008`, `SV-RND-014`, `SV-STY-006`) carry pre-existing compile errors (`compileErrors = 1`) that `FixPlanner`'s compile-error guard leaves unmodified.

## 6 Discussion and Threats to Validity

**Remaining RTL Lowering Blockers.** The 16-repository census (Table 4) shows why front-end elaboration and word-level formal/synthesis lowering belong in separate crates: 99.2% of valid standalone tops pass parsing and elaboration. Under strict flags, 71.1% of unsupported lowering diagnostics stem from just two simulation constructs in `opentitan` and `common_cells`: non-constant immediate assertions inside `initial` blocks (`` `ASSERT_INIT_NET ``, blocking 215 tops and acting as sole blocker on 203) and four-state `$isunknown` checks (blocking 108 tops, sole blocker on 108). Among the 102 stricter rejections, the largest group (21 tops) involves physical time-literal parameters (`parameter int unsigned Delay = 300ps` in `tc_clk.sv`), which require timescale-scaled integer conversion during constant folding. Ranking census diagnostics by *only-blocker* count shows directly how many tops each lowering change unlocks.

**Sound Incompleteness vs. SMT Solving.** For constrained-random static analysis (§4.1), `svlang-lint` trades completeness for predictable polynomial runtime and zero false positives. Non-linear arithmetic ($x * y == 17$), guarded implications (`mode == FAST -> len < 4`), and bit-slice constraints are left unconstrained rather than approximated. Because dropping constraints can only enlarge the solution space, any contradiction proved on the interval or difference-bound projection ($I_{\text{hard}}(x) = \emptyset$, $\sum_{e \in C} w(e) \geq 1$, or $|A| > |\mathcal{D}(T)|$) holds in the

Table 6: RQ4 root-cause breakdown of all 170 ERROR-severity false positives eliminated on OpenTitan's shared DV libraries (~7,900 NBLOC vendored in `ibex-dv`) and 8 ERRORs eliminated on Accellera `uvm-core` through semantic resolution.

| Rule ID & Name | FP Count | Production Idiom Triggering Syntactic FP | Semantic Resolution in `svlang` / `svlang-lint` |
|---|---|---|---|
| *OpenTitan Shared DV Libraries (`dv_lib`, `dv_utils`, `csr_utils`, `dv_base_reg`, `push_pull_agent`, `mem_model`:* 170 → 0 *ERRORs)* | | | |
| `SV-DED-003` `AccidentalEmptyCond` | 78 | ‘`DV_CHECK` expands to `if (T_) ; else ...` so 4-state `X` enters `else` | Empty `then`-branch paired with an explicit `else` branch recognized as deliberate X-safe guard |
| `SV-OOP-011` `DowncastNoDynCast` | 39 | `uvm_radix_enum'(x)`, `uvm_reg_data_t'(y)` matched `uvm_*` prefix | Declaration lookup distinguishes enum/vector `typedef`s from declared `class` types (142 fixed in full `ibex-dv`) |
| `SVA-VAC-001` `SvaVacuousAntecedent` | 17 | `valid && !ready` treated as contradictory boolean conjunction | Requires syntactic/semantic term equivalence between atom and negated atom (`req && !req`) |
| `SV-ERR-002` `UseAfterFailedNull` | 13 | ‘`downcast`/‘`dv_fatal` guards fatal via `uvm_report_enabled(..., $sformatf("%m"))` | Control-flow guard recognizer looks through formatting/name calls (`$sformatf`, ‘`gfn`) on report-ID args |
| `UVM-CFG-005` `ConfigDbCrossComp` | 8 | `dv_base_agent::CFG_T` compared textually with concrete `*_agent_cfg` | Type parameters and local class typedefs excluded from unspecialized cross-component mismatch errors |
| `SV-LANG-005` `NonBlockingInFunc` | 8 | `reset_signals()` drives virtual interface pins via <= (IEEE 1800 §13.4.4) | Declaration-resolved virtual interface and clocking-block targets permitted (aligning with `SV-INT-001`) |
| `SV-INT-001` `BlockingAssignToIfc` | 4 | `cfg.vif.if_mode = ...` static mode setup in `build_phase` | Zero-time pre-simulation setup phases (`build`, `connect`, `end_of_elaboration`, `start_of_simulation`) exempted |
| `UVM-CFG-001` `ConfigDbTypeMismatch` | 2 | `uvm_config_db#(CFG_T)::get` inside parameterized base class | Unspecialized type parameters on either `set` or `get` side exempted consistently with `UVM-CFG-005` |
| `UVM-TST-001` `CheckerCannotFail` | 1 | `dv_base_scoreboard#(RAL_T,CFG_T,COV_T)` abstract base | Parameterized base scoreboards delegate error-reporting checks to concrete subclasses |
| *Accellera `uvm-core` Reference Library (170 files:* 8 → 0 *Objection ERRORs)* | | | |
| `UVM-OBJ-001/003`, `UVM-PHS-006` | 8 | Cross-routine raise/drop pairs, null-guarded raises & loop returns | Unified class-lineage `sem::objections` summary + per-handle symmetric guard tracking (8 → 0 ERRORs) |

full constraint block as well, while keeping analysis fast enough for live `dv-lsp` editing.

**Threats to Validity.** Three limitations apply to the evaluation:

1. *Construct validity on paired rule fixtures (RQ3):* The 415 `(bad.sv, good.sv)` pairs in `testdata/rules/` were built as regression tests for `svlang-lint`'s catalog and follow its rule definitions. To keep the comparison fair, `eval_baselines_on_rules.py` credits `Verible`, `Verilator`, and `Icarus Verilog` whenever any substantive warning or error separates `bad.sv` from `good.sv`, while suppressing fixture-scaffolding warnings on `good.sv`. Testing additional syntax-tree linters such as `dalance/svlint` [8] and third-party bug-injection suites remains useful future work.
2. *Standalone top compilation without project build generators (RQ2):* `census.py` resolves module files without invoking FuseSoC, Bazel, or Makefile code generators, leaving build-generated headers and `+define+` switches unset. Consequently, the 68.7% strict and 86.5% relaxed lowering figures represent conservative lower bounds relative to full project build scripts.
3. *Run-level disk cache invalidation (RQ4):* Because `sv-semantic` builds a cross-file `Design` model, `.svlang_lint_cache/native/` replays runs only when all read source and include files match (14.5×–24.9× warm speedups on unchanged source closures or waiver/config edits). Single-file edits during interactive development rely on `dv-lsp`'s in-memory session rather than partial on-disk cache merges.

Table 7: RQ4 verification lint speed, `.svlang_lint_cache` warm replay speedup, and CST `FixPlanner` verification (`eval_autofix_and_cache.py`, median of 5 release runs on Apple Silicon).

| Benchmark / Metric | Result | Rate / Outcome |
|---|---|---|
| *Accellera `uvm-core` Lint Wall Time (170 files, 86,476 lines)* | | |
| `svlang-lint` (default preset, full elab) | 1.43 s | 60,312 lines/s |
| `svlang-lint --preset all` (all 433 rules) | 1.48 s | 58,568 lines/s |
| `Verilator 5.052 --lint-only --timing` | 1.41 s | 61,527 lines/s |
| `Verible v0.0-4053` (unelaborated CST) | 0.27 s | No include/type elab |
| *Cached Run Replay via `.svlang_lint_cache` (Cold → Warm)* | | |
| Accellera `uvm-core` (`uvm_pkg.sv` closure) | 1.44 s → 0.061 s | **23.7**× speedup |
| 80-package DV suite (2,000 `rand` classes) | 0.20 s → 0.008 s | **24.9**× speedup |
| 80 standalone rule compilation units | 0.14 s → 0.010 s | **14.5**× speedup |
| *CST `FixPlanner` Verification (35 paired autofix rule fixtures)* | | |
| Modified files with 0 post-fix compile errors | 30 / 30 | **100.0%** compile-clean |
| Single-run fixed-point convergence | 29 / 30 | **96.7%** (15 `Safe`, 14 `Unsafe`) |
| Modified files clearing all target findings | 27 / 30 | **90.0%** zero target findings |
| Unmodified by `DisplayOnly` / compile guard | 5 / 35 | 1 `DisplayOnly`, 4 broken |

## 7 Related Work

**Open-Source SystemVerilog Front Ends and Hardware IRs.** Popoloski's `Slang` [12] is the most complete open-source C++17/20 SystemVerilog front end and serves as the differential reference oracle in §3. Relative to `Slang`, `svlang` is written in safe Rust with zero external crates in its core compiler pipeline, retains lossless byte-exact CSTs with dual-origin macro provenance and a preprocessor event table for source-level autofixes, and includes word-level RTL and Annex F SVA automaton lowering (`sv-netlist`) with Btor2, AIGER, and Yosys writers. CHIPS Alliance `Surelog` and `UHDM` [13] pair an ANTLR4 C++ parser with a Cap'n Proto elaborated model aimed at synthesis flows, where serialization overhead makes sub-100 ms editor indexing or browser WebAssembly builds (`svlang-wasm`) impractical. Schuiki et al.'s `LLHD` [21] (and CIRCT/Moore) define multi-level hardware IRs for simulation and synthesis, whereas `svlang` combines IEEE 1800-2023 OOP/UVM specialization typing with compact word-level formal netlists.

**Syntax-Tree Linters and Simulator Lint Modes.** CHIPS Alliance `Verible` [7] (Bison CST) and Hatta's `dalance/svlint` [8] (built on the `nom`-based `sv-parser` IEEE 1800-2017 syntax tree) offer fast style and structural checks, with `dalance/svlint` demonstrating how cleanly single-binary Rust CLIs fit hardware workflows. Because neither constructs a symbol table, folds four-state parameters, elaborates hierarchies, or resolves OOP inheritance and expression widths, they cannot check contradicted constraints, cross-routine UVM objections, or `uvm_config_db` types. On the RTL side, `Verilator`'s `--lint-only` mode [9], `Icarus Verilog` [10], and `Yosys` [11] catch bit-width and synthesis errors well, yet `Verible`, `Verilator`, and `Icarus` together still miss 70.6% of the verification, methodology, constraint, and CDC defects in Table 5.

**Commercial Verification Linters and Industrial Static Analysis.** In commercial EDA, AMIQ Verissimo SystemVerilog Testbench Linter is widely used for UVM, OOP, and constraint checks, while Synopsys VC SpyGlass, Siemens Questa Lint, and RealIntent AscentLint cover RTL, CDC [19], and FSM sign-off. `svlang-lint` brings combined testbench and RTL semantic checking, Verissimo-compatible block waivers (`verissimo_lint: waive_start/waive_stop`), SARIF 2.1.0 [20] reporting, and CST autofixes to open-source silicon projects under Apache-2.0. Driving false-positive `ERROR`s to zero on OpenTitan [1] follows the precision-first engineering discipline established for Coverity by Bessey et al. [14].

## 8 Conclusion

This paper presented `svlang` and `svlang-lint`, a 143,840-line Rust workspace providing an IEEE 1800-2023 SystemVerilog front end, a word-level RTL and Annex F SVA lowerer, a parallel semantic static analyzer, a provenance-guarded CST repair engine, and an LSP server. Verifying each compiler stage against C++ `Slang` through ten differential gates yields 100.0% (1,024/1,024) conformance on `sv-tests`, 99,712/99,712 expression-type agreement, 172/172 UVM 2020.3.1 specialization parity, and 48,834 lines/s RTL lowering across 1,081 open-source top modules. Built on this elaborated representation, `svlang-lint` closes the 70.6% defect-detection gap left by `Verible`, `Verilator`, and `Icarus Verilog` across 415 paired fixtures while reducing `ERROR` false positives to zero on OpenTitan's shared DV libraries and Accellera `uvm-core`.

## Declarations


- **Funding:** Not applicable. This research received no external funding, grants, or institutional support.
- **Competing interests:** The author declares no competing financial or non-financial interests.
- **Ethics approval and consent to participate:** Not applicable.
- **Code and data availability:** The `svlang` compiler SDK, `svlang-lint` static analyzer, `dv-lsp` language server, differential parity gates, benchmark scripts, and 415-pair rule regression suites (release `0.2.0`, commit `ba86329`) are openly available under the Apache-2.0 license at `https://github.com/mayurkubavat/svlang`, published on `crates.io` at `https://crates.io/crates/svlang` and `https://crates.io/crates/svlang-lint`, and on PyPI at `https://pypi.org/project/`

pysvlang/ and https://pypi.org/project/svlang-lint/.

- **Author contributions:** M.K. conceived the compiler and static-analysis architecture, implemented the svlang and svlang-lint infrastructure and language server, designed and executed the differential verification gates and empirical benchmarks, prepared all figures and tables, and wrote and reviewed the manuscript.

- **Use of AI tools:** AI-assisted coding and copy-editing tools were used under the author's supervision to help with boilerplate scaffolding, test-fixture setup, and prose polishing. Every compiler pass, static-analysis rule, differential parity gate, benchmark measurement, and technical statement in this article was designed, executed, and verified by M.K., who assumes full responsibility for the manuscript.

## References


[1] lowRISC Contributors. OpenTitan: Open Source Silicon Root of Trust. https://github.com/lowRISC/opentitan, 2024. Commit 0d2b191a4a.

[2] CHIPS Alliance. Caliptra: Open-Source Root of Trust RTL (caliptra-rtl). https://github.com/chipsalliance/caliptra-rtl, 2024. Commit f2d4641cfa.

[3] OpenHW Group. CORE-V Verification Testbench (core-v-verif). https://github.com/openhwgroup/core-v-verif, 2024.

[4] IEEE. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2023, 2023. DOI: 10.1109/IEEESTD.2024.10458102.

[5] IEEE. IEEE Standard for Universal Verification Methodology Language Reference Manual. IEEE Std 1800.2-2020, 2020. DOI: 10.1109/IEEESTD.2020.9195920.

[6] Microsoft Corporation. Language Server Protocol Specification, Version 3.17. https://microsoft.github.io/language-server-protocol/, 2024.

[7] CHIPS Alliance. Verible: SystemVerilog Developer Tools. https://github.com/chipsalliance/verible, 2024. Version v0.0-4053.

[8] Naoya Hatta. svlint and sv-parser: SystemVerilog Linter and IEEE 1800-2017 Syntax Parser in Rust. https://github.com/dalance/svlint, 2024.

[9] Wilson Snyder. Verilator: Open Simulation—Growing Up. In *DVCon US Proceedings*, 2020. Evaluated with Verilator 5.052, https://www.veripool.org/verilator/.

[10] Stephen Williams. Icarus Verilog. https://github.com/steveicarus/iverilog, 2025. Version 13.0.

[11] Clifford Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/, 2013.

[12] Michael Popoloski. slang: SystemVerilog Language Services. https://github.com/MikePopoloski/slang, 2024. Version 12.1.0.

[13] Alain Dargelas and CHIPS Alliance. Surelog and UHDM: SystemVerilog 2017 Pre-processor, Parser, Elaborator, and Universal Hardware Data Model. https://github.com/chipsalliance/Surelog, 2024.

[14] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, 2010.

[15] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, BtorMC and Boolector 3.0. In *Computer Aided Verification (CAV)*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

[16] CHIPS Alliance. sv-tests: SystemVerilog Compliance Test Suite. https://github.com/chipsalliance/sv-tests, 2024. Commit 415ba5f934ef.

[17] Mayur Kubavat. InFormal: Pure-Rust Formal Property Verification and Equivalence Checking Built on Word-Level RTL Lowering. https://github.com/mayurkubavat/informal, 2026. Working manuscript.

[18] Clifford E. Cummings. Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill! In *Synopsys Users Group Conference (SNUG)*, 2000.

[19] Clifford E. Cummings. Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog. In *Synopsys Users Group Conference (SNUG)*, 2008.

[20] OASIS. Static Analysis Results Interchange Format (SARIF) Version 2.1.0. OASIS Standard, 2020. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[21] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. LLHD: A Multi-level Intermediate Representation for Hardware Description Languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–271. ACM, 2020.