BEHAVE: FUNCTIONAL BEHAVIOR MODELING ENABLES SELF-IMPROVING AGENTS FOR HARDWARE DESIGN AND VERIFICATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Developing agents for hardware design and verification requires reliable correctness feedback. However, a hardware specification may permit correct implementations with different latencies. Matching outputs cycle by cycle to a reference implementation can therefore reject valid designs. To address this, we introduce BEHAVE, an agentic framework for multi-turn joint hardware design and verification through *functional behavior modeling*. We define *Behavior IR* to express task functionality as executable *behavior models* without prescribing implementation timing beyond the specification. The agent develops a register-transfer-level (RTL) design and a behavior model as the design's verification reference. Our evaluator, BEHAVE-Sim, generates input stimuli and checks both artifacts separately against a golden behavior model, using goal-guided search to exercise additional execution cases. BEHAVE thus supports power, performance, and area (PPA) exploration across task-permitted latencies and microarchitectures. During training, the same evaluator provides verifiable reinforcement learning (RL) rewards from specification-behavior pairs without reference RTL. For self-improvement, we show that the agent can search for high-level implementations relevant to its capability gaps, construct and check specification-behavior pairs, and continue training. We release BEHAVE-Train and BEHAVE-Eval with 600 human-reviewed specification-behavior pairs for realistic hardware workloads. On BEHAVE-Eval, self-improvement from 60 seed tasks raises Qwen3.8-27B's RTL pass@1 from 55.0% to 75.0%, reaching performance comparable to RL using a 540-task pool.

1 INTRODUCTION

Design and verification are key stages of digital chip development (Rabaey et al., 2003; Bergeron, 2003). Figure 1(a) illustrates a common workflow: engineers develop a register-transfer-level (RTL) design and a reference model for verification, with a testbench driving both and comparing their outputs (IEEE, 2020). Output mismatches help engineers debug the design or reference model (Wile et al., 2005). Large language model (LLM) agents (Yang et al., 2024; Wei et al., 2025b) open a path to automating this process. We study how to develop an agent that jointly generates RTL designs and reference models for verification during inference and learns both capabilities during training.

Developing such an agent requires reliable correctness feedback for both its RTL designs and reference models. However, matching RTL outputs cycle by cycle to a reference implementation ties this feedback to that implementation's timing (Yu et al., 2026b; Ye et al., 2025a; Mu et al., 2025; Tan et al., 2026b; Zhao et al., 2025b). When the specification permits different latencies, this can reject valid designs with preferable power, performance, and area (PPA) trade-offs, as illustrated in Figure 1(b). The challenge is therefore to provide executable supervision that checks task-defined behavior while preserving the timing flexibility allowed by the specification.

We address this challenge through *functional behavior modeling*, using executable *behavior models* to describe task functionality (Figure 1(c)). We express them in *Behavior IR*, a typed representation. One behavior model can serve as a reference for RTL implementations with task-permitted differences in latency and microarchitecture. For a specification S, the agent develops an RTL design $\hat{R}$ and a

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

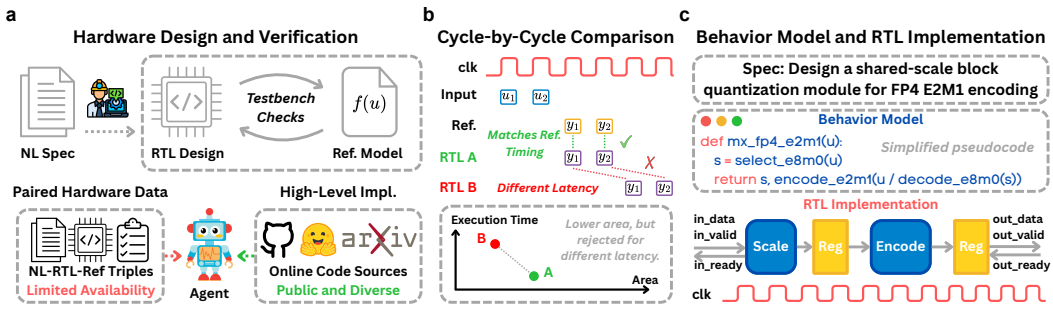


Figure 1: **Motivation for functional behavior modeling.** (a) A common development workflow and the availability gap between paired hardware data and public high-level implementations. (b) Cycle-by-cycle comparison rejects a correct RTL design with task-permitted latency differences, despite its lower area. (c) A behavior model and pipelined RTL for MXFP4 block quantization. See Appendix A.2 for task details and code excerpts of the Python behavior model and pipelined RTL.

behavior model $\hat{B}$ as the design's verification reference. Our evaluator, BEHAVE-Sim, independently checks both artifacts against a *golden behavior model* $B^\star$ following task-defined rules.

Building on this, we develop BEHAVE, an agentic framework for multi-turn joint design and verification. The agent iteratively refines $\hat{R}$ and $\hat{B}$ using its own testbench and development tools, and can use EDA feedback to explore area-time trade-offs. For independent evaluation, BEHAVE-Sim first runs random tests. It then uses solver-guided input search to target unhit *goals*, such as branch outcomes or assertion violations, using feedback from both $B^\star$ and the submitted artifact.

During training, BEHAVE-Sim's artifact scores provide verifiable rewards for reinforcement learning (RL). The agent can thus learn RTL design and reference-model construction from $(S, B^\star)$ pairs without reference RTL. While RTL training data are scarce (Thakur et al., 2024), high-level implementations are widely available online (Kocetkov et al., 2023), as shown in Figure 1(a). To acquire new training tasks, the agent searches for implementations relevant to its capability gaps and adapts them into behavior models with corresponding specifications. After checking these pairs, it trains on them and uses feedback from the updated agent to guide subsequent searches, supporting continued self-improvement on realistic hardware workloads. Our contributions are:

- **Functional behavior modeling.** We develop BEHAVE, an agentic framework for multi-turn co-development of RTL designs and behavior models for verification. The agent iteratively refines both artifacts using tool feedback and explores area-time trade-offs with EDA tools. BEHAVE-Sim evaluates both submitted artifacts against golden behavior models.
- **Verifiable training and self-improvement.** We enable verifiable RL from specification-behavior pairs derived from a growing body of public high-level implementations, without requiring reference RTL. The agent alternates capability-guided task acquisition with training, supporting continued self-improvement on evolving workloads such as AI kernels.
- **Open-source datasets.** We release BEHAVE-Train and BEHAVE-Eval, comprising 600 human-reviewed specification-behavior pairs covering realistic hardware workloads in AI/ML, numerics, signal processing, cryptography, control, compression, and data systems.

2 Background

This section reviews RTL design and simulation-based verification in Subsection 2.1, then explains the roles of reference models and testbenches in Subsection 2.2.

2.1 Hardware Design and Verification

RTL Design. In digital circuit design, an RTL design (e.g., in Verilog or SystemVerilog) specifies combinational logic and sequential state updates, typically governed by clocks and resets. Unlike an

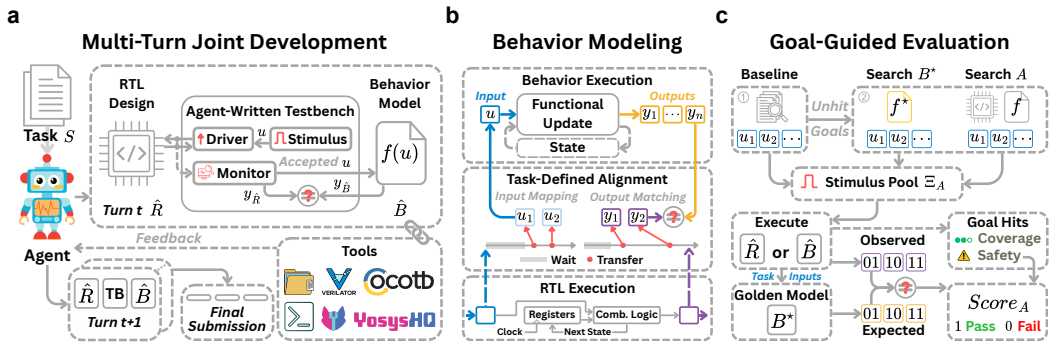


Figure 2: **Joint development and evaluation.** (a) The agent iteratively develops RTL $\hat{R}$, a behavior model $\hat{B}$, and a testbench through tool feedback. (b) Task-defined alignment matches RTL outputs to behavior-model predictions from accepted inputs, allowing task-permitted latency differences. (c) For each submitted artifact $A \in \{\hat{R}, \hat{B}\}$, baseline tests and solver-guided searches on $B^\star$ and A provide stimuli for independent evaluation against $B^\star$, checking output agreement and safety-goal hits.

untimed high-level implementation (e.g., in Python), RTL explicitly describes how computation and data transfers are organized across clock cycles (Rabaey et al., 2003).

RTL Verification Step 1: Stimulus Generation. Simulation-based verification first generates input stimuli to exercise the RTL design under test (DUT). Directed testing uses manually selected input sequences to target known scenarios, constrained-random testing samples a broader input space under user-defined constraints (Naveh et al., 2007), and concolic testing combines concrete execution with symbolic constraint solving to steer inputs toward unexplored paths (Sen et al., 2005).

RTL Verification Step 2: Correctness Checking. The DUT's responses to these stimuli are checked for correctness. Outputs may be compared with expected values specified directly or computed by a reference model on the same inputs (Bergeron, 2003). Beyond output-value comparison, assertions can check protocol and temporal properties during execution (Wile et al., 2005).

2.2 Reference Models and Testbenches

Reference Models and High-Level Implementations. A reference model is an executable description of the DUT's expected behavior. Reference models vary in language and range from cycle-accurate to untimed (Bergeron, 2003). For a high-level implementation to serve as a reference model, details such as bit widths, overflow behavior, and state initialization must be explicit and consistent with the task specification. In our implementation, reference models are expressed in Behavior IR, a typed executable representation with defined semantics and a restricted set of constructs.

UVM Testbenches. A testbench coordinates the stimulus generation and correctness checking described above, as in Figure 2(a). The Universal Verification Methodology (UVM) provides a standardized framework for organizing these functions into components (IEEE, 2020). *Sequences* generate stimulus transactions, *drivers* translate them into DUT interface signals, and *monitors* record the DUT's input and output transactions. *Scoreboards* compare observed outputs with reference-model predictions for the recorded inputs, while *coverage collectors* track exercised scenarios.

3 BEHAVE

We first describe functional behavior modeling for multi-turn joint design and verification (Section 3.1). We then present BEHAVE-Sim's goal-guided evaluation (Section 3.2). Finally, we describe PPA exploration (Section 3.3) and verifiable RL with search-based self-improvement (Section 3.4).

3.1 Functional Behavior Modeling

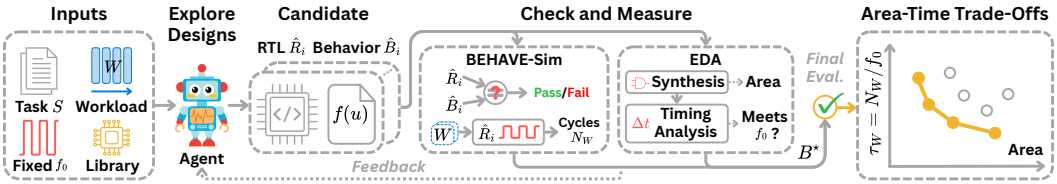


Figure 3: **Timing-flexible PPA exploration.** The agent freely explores area-time trade-offs through multi-turn tool interaction. Orange points mark the empirical Pareto frontier.

Task setup. Each task pairs a natural-language (NL) specification S with a golden behavior model $B^\star$. The agent receives S, while $B^\star$ is reserved for evaluation and remains hidden during development. The agent develops an RTL design $\hat{R}$ and a behavior model $\hat{B}$ as the design's verification reference.

Multi-turn development. As shown in Figure 2(a), the agent iteratively edits and tests $\hat{R}$ and $\hat{B}$ with development tools, using its own testbench to check their agreement and guide revisions. After the agent submits its final $\hat{R}$ and $\hat{B}$, BEHAVE-Sim evaluates each independently against $B^\star$.

Behavior models. The golden model $B^\star$ and the agent-written model $\hat{B}$ are represented in Behavior IR, which supports typed operations, arrays, conditionals, loops, and persistent state. Each invocation takes task-defined inputs and state, producing updated state and ordered outputs or an explicit failure. Appendix A.2 provides an MXFP4 example. Appendix B.7 gives the syntax and semantics.

Task-defined alignment. For RTL-model comparison, $\hat{R}$ is checked against $\hat{B}$ during development and $B^\star$ during evaluation. As shown in Figure 2(b), the reference behavior model processes the inputs accepted by the RTL and task-defined events such as reset. Its predicted outputs are matched to RTL outputs, allowing latency differences permitted by the task. For model-model comparison, $\hat{B}$ and $B^\star$ receive the same task-defined inputs, and their outputs are compared directly.

Ready/valid example. Consider a ready/valid task that requires one result per accepted input but leaves response latency unspecified. BEHAVE-Sim records the RTL's input and output data at clock edges when the data are marked valid and the receiver is ready to accept them. It feeds the recorded input values to the reference behavior model and compares its predictions with the recorded RTL outputs. Appendix A.4 summarizes supported protocols and limitations.

3.2 Goal-Guided Evaluation

Evaluation setup. BEHAVE-Sim evaluates each submitted artifact $A \in \{\hat{R}, \hat{B}\}$ on a finite stimulus pool Ξ_A, comparing its outputs with $B^\star$. Each stimulus $\xi \in \Xi_A$ consists of finite input sequences and an execution schedule specifying input presentation, clock events, and resets.

Execution goals. A goal is hit when its named event occurs. Coverage goals mark execution cases, such as branch outcomes, while safety goals mark failures, such as violations of existing assertions. BEHAVE-Sim automatically extracts goals from artifact structure and predefined protocol checks, while tasks can explicitly declare additional goals. Appendix A.6 details goal types and hit conditions.

Stimulus generation. As shown in Figure 2(c), BEHAVE-Sim first runs edge-case and seeded random tests. Two searches then use execution feedback from $B^\star$ and A, respectively, to generate inputs targeting unhit goals. Each search targets goals whose triggering conditions can be encoded as solver constraints on input values. Solver-generated tests are executed to check outputs and update goal hits, guiding further search within a shared time budget. Appendix A.7 gives the full procedure.

Artifact score. An artifact passes on Ξ_A when its outputs match $B^\star$ and it hits no safety goal:

$$\mathrm{score}_A(\Xi_A) = \mathbf{1}\left[n^A_{\mathrm{mis}}(\Xi_A) = 0 \wedge \mathrm{Hit}_A(\Xi_A) \cap \Gamma^{\mathrm{safe}}_A = \emptyset\right],$$

where n^A_{mis} counts output mismatches, Hit_A collects observed goal hits, and Γ^{safe}_A is A's safety-goal set. Goal coverage is reported separately and does not affect this score (Appendix A.6).

Theorem 1 (Bounded goal analysis)**.** *Under the assumptions in Appendix B, for either submitted artifact, every goal reported as reached has an execution that hits it, confirmed by replay. A goal*

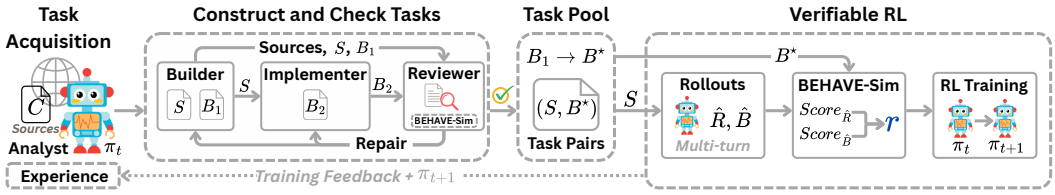


Figure 4: **Task acquisition and verifiable RL.** All roles use the current policy π_t in separate contexts. Accepted task pairs support RL, and training feedback guides subsequent task acquisition.

reported as bounded-unreachable cannot be hit by any allowed execution within the fixed bounds. No conclusion is drawn for unresolved goals. This analysis does not change the artifact score.

Proof. Apply Theorem 3 to the exact bounded encodings in Theorems 4 and 5 (Appendix B). □

3.3 Timing-Flexible PPA Exploration

Candidate search. Given S, a fixed workload W, frequency f_0, and cell library, the agent seeks low area and short workload execution time. It can vary parallelism and other choices when revising $\hat{R}$ and $\hat{B}$ with BEHAVE-Sim and EDA feedback (Figure 3). BEHAVE-Sim checks their agreement and measures workload cycles, while synthesis and timing analysis report cell area and timing.

Area-time trade-offs. Workload execution time is $\tau_W = N_W / f_0$, where N_W counts RTL cycles to complete W. Finally, RTL candidates are evaluated against the hidden reference $B^\star$. We report all passing candidates that complete W and meet timing at f_0, highlighting the empirical Pareto frontier.

3.4 Verifiable Reinforcement Learning and Self-Improvement

Verifiable RL. Given $(S, B^\star)$ pairs, we train the agent with GRPO (Shao et al., 2024) on multi-turn joint-development rollouts. BEHAVE-Sim evaluates $\hat{R}$ and $\hat{B}$ against $B^\star$, yielding the rollout reward

$$r = \frac{\text{score}_{\hat{B}}(\Xi_{\hat{B}}) + \text{score}_{\hat{R}}(\Xi_{\hat{R}})}{2} \in \left\{0, \frac{1}{2}, 1\right\}.$$

Feedback-guided task acquisition. To expand the training set, the current policy π_t acts as an Analyst (Figure 4). It reviews completed training rollouts and their feedback to identify capability gaps, then selects relevant high-level implementations and sets objectives for new tasks. The Analyst records useful findings as experience to guide subsequent acquisition rounds.

Constructing training pairs. Builder constructs S and B_1 from the sources and objectives. In a separate context, Implementer produces B_2 from S and public interface documentation, without the sources or B_1. Reviewer checks the task against its sources and compares the models with BEHAVE-Sim, requesting repairs when needed. Accepted (S, B_1) pairs enter the cumulative training pool with B_1 as $B^\star$. Training continues from π_t, with π_{t+1} and rollout feedback guiding acquisition.

4 Experiments and Discussion

We first introduce the datasets and evaluation protocol (Section 4.1), then examine how well models jointly perform hardware design and verification using BEHAVE (Section 4.2). We compare BEHAVE with other development pipelines and test whether providing BEHAVE-Sim as a stimulus-generation tool improves success and reduces cost (Section 4.3). We then test whether agents can explore area-time trade-offs under a fixed workload and frequency (Section 4.4), and whether verifiable RL and continued task acquisition improve their capabilities (Section 4.5).

4.1 Benchmarks and Evaluation Settings

Benchmarks. We construct 600 human-reviewed specification-behavior pairs from high-level implementations across seven application domains, split into BEHAVE-Train (540 tasks) and BEHAVE-

Table 1: **Main results on hardware design benchmarks (%).** RTL and Behav. report RTL and Behavior pass@1 under BEHAVE-Sim. Cov. is line coverage achieved by the submitted testbench on the submitted RTL, averaged over all tasks. Missing submissions or compilation/execution failures score zero. RL and SI denote fixed-dataset RL and self-improvement (see Section 4.5).

	VerilogEval-v2			RTLLM-v2			CVDP-cid003			BEHAVE-Eval		
Model	RTL	Behav.	Cov.	RTL	Behav.	Cov.	RTL	Behav.	Cov.	RTL	Behav.	Cov.
Qwen3.5-4B	34.6	28.8	6.9	12.0	20.0	0.0	4.3	4.3	0.9	0.0	6.7	0.0
Qwen3.5-9B	46.8	40.4	13.4	18.0	22.0	2.1	7.0	9.6	2.4	0.0	15.0	1.8
Claude Haiku 4.5	92.9	89.1	62.7	88.0	84.0	87.2	72.2	54.8	83.8	36.7	70.0	48.1
GPT-5.6 Luna	94.2	94.9	84.1	98.0	98.0	85.3	93.0	93.9	91.6	68.3	73.3	89.7
Claude Sonnet 5	92.3	92.3	58.1	84.0	86.0	75.7	64.3	60.9	66.9	68.3	65.0	60.7
GPT-5.6 Terra	99.4	99.4	71.0	98.0	98.0	85.6	95.7	97.4	95.9	85.0	90.0	95.7
DeepSeek-V4.1-Flash	96.8	96.2	79.9	98.0	100.0	94.0	93.0	90.4	95.5	88.3	83.3	90.7
Qwen3.8-27B	97.4	95.5	77.3	90.0	90.0	87.9	81.7	82.6	87.0	55.0	56.7	59.1
Qwen3.8-27B-RL (ours)	98.1	96.8	66.7	96.0	92.0	85.7	91.3	87.8	92.0	73.3	71.7	80.6
Qwen3.8-27B-SI (ours)	98.7	96.2	80.6	96.0	92.0	85.8	93.0	92.2	91.5	75.0	75.0	77.5

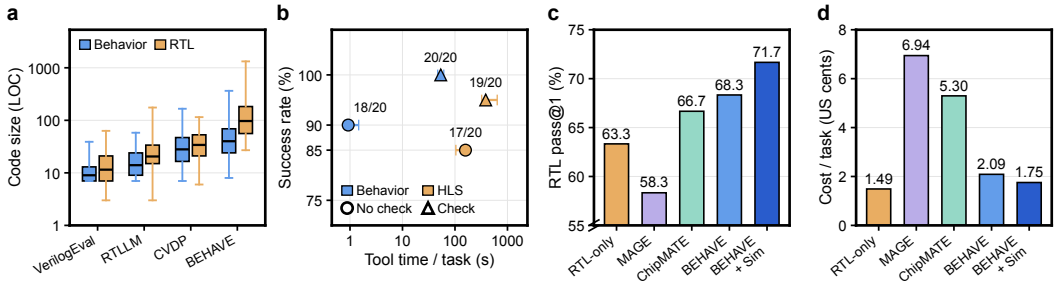


Figure 5: **Benchmark construction and pipeline comparison.** (a) RTL and Behavior code sizes across benchmarks. (b) End-to-end success and median tool time (bars: interquartile range) for Behavior IR and HLS on 20 tasks with GPT-5.6 Luna. Check denotes development-time correctness feedback; both retain frontend/synthesis feedback and final GT evaluation. (c) RTL pass@1 across pipelines on BEHAVE-Eval. (d) Mean API cost for the same pipelines.

Eval (60 tasks). We extend VerilogEval-v2 (Pinckney et al., 2025a), RTLLM-v2 (Liu et al., 2024c), and CVDP-cid003 (Pinckney et al., 2025b) with golden Behavior models and resolve specification inconsistencies. Behavior models typically use less code than RTL by omitting hardware implementation details. BEHAVE has a larger median RTL code size than the other benchmarks (Figure 5(a)). Appendix D details sources, construction, review, splits, and licensing.

Behavior IR vs. HLS. High-level synthesis (HLS) converts designs written in high-level languages, such as C++, into RTL. On 20 tasks, the agent translates the same high-level implementations into Behavior IR or synthesizable C++. BEHAVE-Sim and Vitis HLS provide compilation and synthesis feedback, respectively. With or without BEHAVE-Sim correctness feedback, Behavior IR achieves higher success rates and lower median tool time (Figure 5(b)). Appendix C.4 provides details. Our framework is not tied to Behavior IR and can use other executable reference representations.

Evaluation settings. Models share a development environment with a 128K-token context window and at most 60 turns per rollout. BEHAVE-Sim evaluates final RTL and Behavior submissions. Appendices C.1 and C.2 detail the environment, model configurations and evaluation budgets.

4.2 Hardware Design and Verification Performance

Model capabilities. Table 1 shows that frontier foundation models already exhibit strong hardware design and verification capabilities. Given a tool-equipped development environment, they can autonomously write, test, and revise RTL designs and their Behavior references, using tool feedback to decide their next steps. Smaller models still struggle with both artifacts, and testbench coverage remains uneven even among models with high RTL and Behavior pass@1.

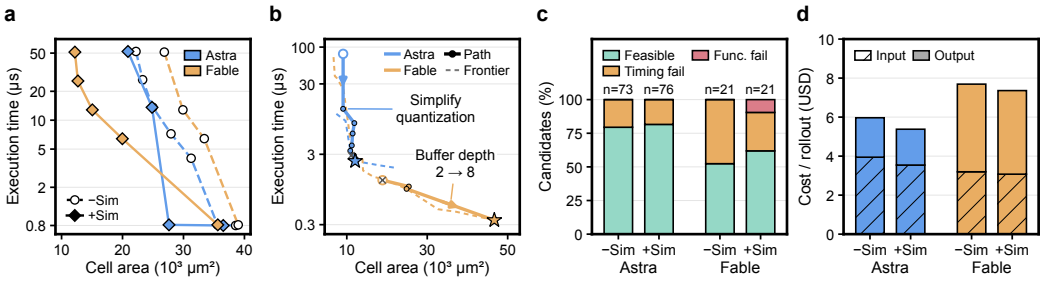


Figure 6: **Timing-flexible PPA exploration.** (a) MXFP4 frontiers with and without BEHAVE-Sim (SkyWater 130 nm, 80 MHz). (b) NVFP4 frontiers (dashed) and optimization paths (solid) (FreePDK45-based Nangate 45 nm library, 400 MHz). Stars mark endpoints and crosses timing failures. (c) MXFP4 candidate outcomes (n: collected candidates). (d) MXFP4 mean estimated input/output API cost per search with cross-turn cache reuse. Execution time is simulated cycles / target frequency for 64 continuously offered blocks of 32 (MXFP4) or 16 (NVFP4) BF16 values.

Benchmark saturation. In our interactive setting, strong models approach saturation on these extended benchmarks, while BEHAVE-Eval remains more challenging. GPT-5.6 Terra achieves 95.7-99.4% RTL pass@1 across the three external benchmarks, compared with 85.0% on BEHAVE-Eval. As algorithms and workloads evolve, we can construct harder tasks from high-level implementations.

4.3 Comparison of Development Workflows

Comparison setup. Using GPT-5.6 Luna, we compare RTL-only development, ChipMATE (Yu et al., 2026b), MAGE (Zhao et al., 2025c), BEHAVE, and BEHAVE with BEHAVE-Sim on BEHAVE-Eval. In the last setting, BEHAVE-Sim generates stimuli and compares the agent's RTL and Behavior model during development, replacing the agent-written testbench. All final RTL submissions are evaluated against golden Behavior models. Appendix C.5 details the settings and baseline adaptations.

Performance and cost. BEHAVE achieves higher RTL pass@1 than RTL-only development, suggesting that constructing an executable reference can help the agent develop RTL (Figure 5(c)). With a single tool-using agent, BEHAVE achieves comparable RTL pass@1 to ChipMATE and MAGE at lower API cost (Figure 5(d)). Adding BEHAVE-Sim improves success while reducing cost, supporting the value of automated testing within joint development.

Why functional behavior modeling. ChipMATE generates Python reference models for RTL verification. Reproducing the RTL's cycle-by-cycle execution in Python, however, can retain its control logic and intermediate state. In an unsigned-division example, ChipMATE's Python model maintains control state across clock-level calls, whereas the Behavior reference completes the computation in one call (see Appendix C.8 for details). Expressing the same computation at different abstraction levels allows joint development to separate functional intent from architectural choices.

4.4 Timing-Flexible PPA Exploration

Experimental setup. We compare BEHAVE with and without BEHAVE-Sim for MXFP4 area-time exploration using GPT-6 Astra and Claude Fable 5.1. Both use EDA feedback under the same workload, clock frequency, and SkyWater 130 nm cell library (SkyWater PDK Authors, 2020). We also apply the same flow to NVFP4 using the FreePDK45-based Nangate 45 nm standard-cell library (Nangate, 2008; Stine et al., 2007). Appendix C.5 details search and measurement settings.

Area-time trade-offs. Both models produce correct designs with different area-time trade-offs (Figure 6(a)). Figure 6(b) traces NVFP4 revisions to selected frontier points. Astra simplifies quantization using comparisons in place of division. Fable increases buffering from two to eight blocks, reducing execution time at the cost of area. Within each task, the same golden Behavior reference validates designs with different architectures and execution times.

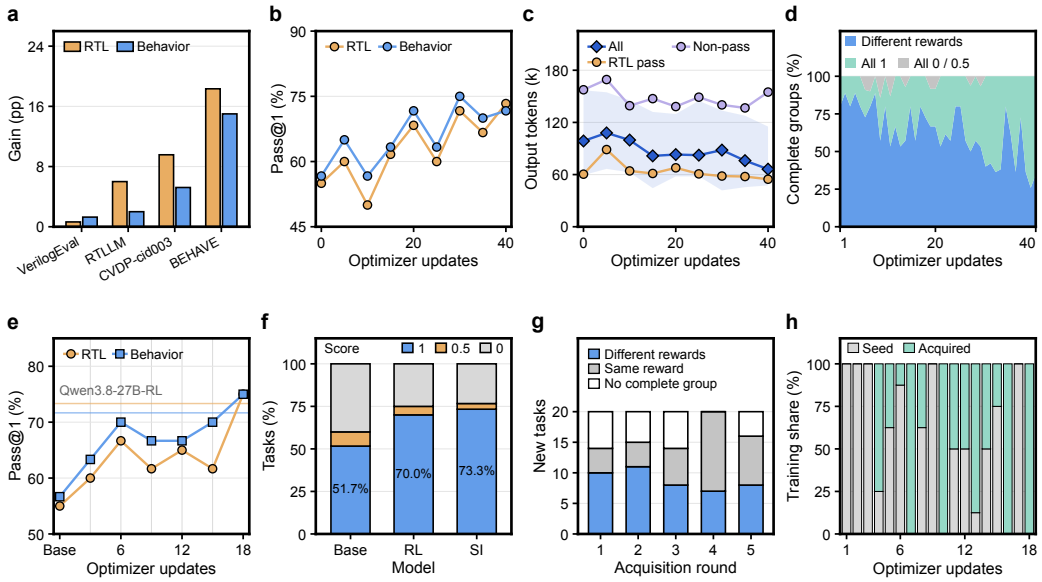


Figure 7: **RL and self-improvement with Qwen3.8-27B.** (a) RL gains across benchmarks. (b,c) BEHAVE-Eval pass@1 and median rollout output length. (d) Reward-group composition during RL. (e) Self-improvement performance on BEHAVE-Eval. (f) Joint-score distributions on BEHAVE-Eval: 1, 0.5, and 0 indicate both, one, or neither artifact passing. (g) Reward variation across eight candidate solutions per newly acquired task. (h) Seed/acquired task shares in each training update.

Effect of simulation feedback. BEHAVE-Sim yields lower-area designs at comparable execution times and a higher fraction of feasible candidates (Figure 6(a,c)). Estimated mean API cost per search is modestly lower with BEHAVE-Sim (Figure 6(d)). These results suggest that reusable stimulus-generation and comparison tools can be more effective than asking agents to write testbenches.

4.5 Verifiable RL and Self-Improvement

Experimental setup. We train Qwen3.8-27B for 40 updates, sampling from the 540-task BEHAVE-Train pool. Self-improvement starts from 60 seed tasks and adds new tasks through 5 acquisition rounds, with 18 training updates in total. Appendix C.6 details the training and acquisition settings.

Learning performance. Verifiable RL improves both RTL design and reference-model construction, with the largest gains on BEHAVE-Eval. RTL pass@1 rises from 55.0% to 73.3%, and Behavior pass@1 from 56.7% to 71.7% (Figure 7(a-b)). These gains are accompanied by shorter generations: median total generation length per rollout falls from 98.6k to 66.2k tokens (Figure 7(c)).

Self-improvement. The final self-improved model reaches 75.0% for both RTL and Behavior pass@1 on BEHAVE-Eval (Figure 7(e)). Both artifacts pass on 44 of 60 tasks, versus 31 before training and 42 after fixed-dataset RL (Figure 7(f)). Starting from 60 seed tasks, self-improvement reaches the performance of RL using a 540-task pool. Appendix C.9 presents a recorded self-improvement case.

Learning signal. Later fixed-dataset updates contain more sampled all-1 groups, leaving no reward differences within those groups (Figure 7(d)). Self-improvement expands the pool with tasks constructed from retrieved software. Newly acquired tasks provide nonconstant rewards across acquisition rounds (Figure 7(g)) and contribute to subsequent updates (Figure 7(h)).

5 Related Work

Verification-Guided Hardware Agents. MAGE generates Verilog testbenches for RTL debugging, ChatModel generates SystemC reference models, and PRO-V-R1 trains agents to generate Python reference models and testbenches using golden RTL supervision (Zhao et al., 2025c; Ye et al., 2025a;

Table 2: **Comparison of related methods.** *Task* lists evaluated capabilities. RM and TB denote reference model and testbench. G and P mark agent-generated verification artifacts and supplied references. *Timing Flex.* indicates whether RTL latency or kernel runtime may vary. *Self-improvement* indicates whether the updated agent autonomously acquires new tasks for further training.

	Inference					Training	
Method	**Task**	**Design**	**Verification**	**Alignment**	**Timing Flex.**	**Training Source**	**Self-Improv.**
MAGE (Zhao et al., 2025c)	Design	RTL	Verilog $TB^{G/P}$	Cycle	No	-	No
ChatModel (Ye et al., 2025a)	Verification	-	SystemC RM^{G}	Cycle	No	-	No
PRO-V-R1 (Zhao et al., 2025b)	Verification	-	Python RM^{G}	Cycle	No	Spec-RTL	No
FAVer (Mu et al., 2025)	Design	RTL	Python RM^{G}	Cycle	No	Spec-RTL	No
AutoVeriFix+ (Tan et al., 2026b)	Both	RTL	Python RM^{G}	Cycle	No	-	No
ChipMATE (Yu et al., 2026b)	Both	RTL	Python RM^{G}	Cycle	No	Spec-RTL	No
CorrectHDL (Xu et al., 2025)	Design	RTL	HLS RTL^{P}	Task-specific	Yes	-	No
FormalRTL (Li et al., 2026)	Design	RTL	C RM^{P}	Fixed-Latency	No	-	No
HINT (Cheng et al., 2026)	Design	RTL	HINT IR^{G}	Task-specific	No	-	No
CUDA Agent (Dai et al., 2026)	Design	CUDA	PyTorch RM^{P}	Completion	Yes	PyTorch Ops	No
DRTriton (Guo et al., 2026)	Design	Triton	PyTorch RM^{P}	Completion	Yes	PyTorch Ops	No
BEHAVE (ours)	**Both**	**RTL**	**Behavior RM^{G}**	**Task-specific**	**Yes**	**Spec-Behavior**	**Yes**

Zhao et al., 2025b). FAVer and AutoVeriFix+ use Python reference models to guide RTL generation and repair (Mu et al., 2025; Tan et al., 2026b). ChipMATE trains RTL and Python agents for cross-verification, with the Python agent supervised by RTL-derived, cycle-accurate reference models (Yu et al., 2026b). Cycle-aligned checking couples verification to reference timing, limiting reuse of the same reference across RTL designs with task-permitted latency differences (Table 2).

High-Level Representations and References. CorrectHDL checks generated RTL against HLS-generated references derived from C/C++ (Xu et al., 2025), while FormalRTL uses C reference models for equivalence checking (Li et al., 2026). HINT generates an executable IR with a frozen transaction and timing contract for RTL verification (Cheng et al., 2026). Yin et al. (2026) use hierarchical IRs to describe module connectivity, operations, and I/O behavior. CUDA Agent and DRTriton compare generated kernels with PyTorch references after kernel completion (Dai et al., 2026; Guo et al., 2026). BEHAVE matches RTL outputs to behavior-model predictions according to task-defined rules, preserving timing flexibility permitted by the specification.

Self-Improving Agents. Agents can refine memory and skills (Zhang et al., 2026a; Yan et al., 2026), evolve agent code and harnesses (Zhang et al., 2026c), or update model weights through training (Zhang et al., 2026d; Wang et al., 2025d). Some training methods let models propose new problems and learn from solving them (Zhao et al., 2025a; Huang et al., 2026; Wang et al., 2025c; Yang et al., 2026; Lu et al., 2026a; Dong et al., 2026). BEHAVE acquires new tasks together with executable references from public high-level implementations. The updated agent guides acquisition using training feedback, and checked specification-behavior pairs support further RL.

For extended related work on LLMs for EDA and hardware benchmarks, see Appendix E.

6 Conclusion

We presented BEHAVE, a framework for joint hardware design and verification through functional behavior modeling. BEHAVE-Sim independently evaluates RTL designs and behavior models against golden references while preserving task-permitted timing flexibility, providing verifiable RL rewards without reference RTL. Experiments demonstrate multi-turn joint development, area-time exploration across workloads and cell libraries, and improvements in RTL and reference-model generation through training. Starting from a small human-reviewed seed set, the agent constructs and checks new specification-behavior pairs for continued training, using rollout feedback to guide task acquisition. This self-improvement process reaches performance comparable to full-dataset RL on BEHAVE-Eval. By turning high-level implementations into executable supervision, BEHAVE offers a path to continued self-improvement as algorithms and workloads evolve.

AI USE STATEMENT

We used generative AI tools to polish the manuscript and improve the presentation of the mathematical claims in Appendix B. These tools also assisted literature retrieval and review for Section 5 and Appendix E. LLMs also assisted the retrieval of high-level source implementations and the construction of specification-behavior pairs, including drafting specifications, generating behavior models and auxiliary RTL, and revising these artifacts using execution feedback. BEHAVE benchmark construction and external-benchmark adaptation are detailed in Appendices D.2 and D.3, respectively. Human review of these benchmark tasks is described in Appendix D.4. The autonomous acquisition of additional training tasks during self-improvement is described separately in Section 3.4 and Appendix C.6. The authors reviewed the AI-assisted contributions and remain responsible for the final text, mathematical claims, code, datasets, experiments, and reported results.

ETHICS STATEMENT

Our benchmark is derived from publicly available technical material and does not intentionally collect personal or sensitive data. We record source provenance and licensing, and will release only human-reviewed benchmark items with established redistribution permissions. Hardware-generation systems are dual-use, and generated designs may contain functional, safety or security defects. The checks in BEHAVE reduce but do not eliminate these risks. Generated designs therefore require independent validation before deployment.

REPRODUCIBILITY STATEMENT

We will publicly release the BEHAVE source code, benchmark, golden Behavior models, auxiliary reference RTL, prompts, training and evaluation configurations, model checkpoints and per-run records under open licenses. Reference RTL is not required for reward computation or autonomous task acquisition. The release records tool and dependency versions, model identifiers, random seeds, run policies and artifact digests needed to reproduce each result. Source-specific provenance, licensing and attribution are documented in Appendix D.

REFERENCES

Manar Abdelatty, Maryam Nouh, Jacob K. Rosenstein, and Sherief Reda. Pluto: A benchmark for evaluating efficiency of LLM-generated hardware code. *arXiv preprint arXiv:2510.14756*, 2025.

Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. Toward an open-source digital flow: First learnings from the OpenROAD project. In *Proceedings of the Design Automation Conference*, 2019.

Mohammad Akyash, Kimia Azar, and Hadi Kamali. Rtl++: Graph-enhanced llm for rtl code generation. In *Proceedings of the IEEE International Conference on LLM-Aided Design*, 2025.

Ahmed Allam and Mohamed Shalan. Rtl-repo: A benchmark for evaluating llms on large-scale rtl design projects. In *Proceedings of the IEEE LLM Aided Design Workshop*, 2024.

AMD. Vitis high-level synthesis user guide (UG1399), version 2025.2, 2026. URL https://docs.amd.com/r/2025.2-English/ug1399-vitis-hls/Introduction.

Arm. AMBA 4 AXI4-Stream protocol specification. Specification IHI 0051A, Arm, 2010. URL https://documentation-service.arm.com/static/642583d7314e245d086bc8c9.

Arm. AMBA AXI and ACE protocol specification. Specification IHI 0022H, Arm, 2020. URL https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/IHI0022H_amba_axi_protocol_spec.pdf.

Arm. AMBA APB protocol specification. Specification IHI 0024E, Arm, 2023. URL https://documentation-service.arm.com/static/63fe2c1356ea36189d4e79f3.

Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.

Jitendra Bhandari, Johann Knechtel, Ramesh Narayanaswamy, Siddharth Garg, and Ramesh Karri. Llm-aided testbench generation and bug detection for finite-state machines. *arXiv preprint arXiv:2406.17132*, 2024.

Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *Proceedings of the International Conference on Computer Aided Verification*, 2010.

Paul E. Calzada, Zahin Ibnat, Tanvir Rahman, Kamal Kandula, Danyu Lu, Sujan Kumar Saha, Farimah Farahmandi, and Mark Tehranipoor. Verilogdb: The largest, highest-quality dataset with a preprocessing framework for llm-based rtl generation. *arXiv preprint arXiv:2507.13369*, 2025.

Fu-Chieh Chang, Yu-Hsin Yang, Hung-Ming Huang, Yun-Chia Hsu, Yin-Yu Lin, Ming-Fang Tsai, Chun-Chih Yang, and Pei-Yuan Wu. Specloop: An agentic rtl-to-specification framework with formal verification feedback loop. *arXiv preprint arXiv:2603.02895*, 2026.

Kaiyan Chang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, Xiaowei Li, and Ying Wang. Improving large language model hardware generating quality through post-llm search. In *Workshop on Machine Learning for Systems at NeurIPS*, 2023.

Hongzheng Chen, Yingheng Wang, Yaohui Cai, Hins Hu, Jiajie Li, Shirley Huang, Chenhui Deng, Rongjian Liang, Shufeng Kong, Haoxing Ren, Samitha Samaranayake, Carla P. Gomes, and Zhiru Zhang. Heurigym: An agentic benchmark for llm-crafted heuristics in combinatorial optimization. In *Proceedings of the International Conference on Learning Representations*, 2026a.

Mu-Chi Chen, Yu-Hung Kao, Po-Hsuan Huang, Shao-Chun Ho, Hsiang-Yu Tsou, I-Ting Wu, En-Ming Huang, Yu-Kai Hung, Wei-Po Hsin, Cheng Liang, Chia-Heng Tu, Shih-Hao Hung, and H. T. Kung. Siliconmind-v1: Multi-agent distillation and debug-reasoning workflows for verilog code generation. *arXiv preprint arXiv:2603.08719*, 2026b.

Zhirong Chen, Kaiyan Chang, Zhuolin Li, Cangyuan Li, Xinyang He, Chujie Chen, Mengdi Wang, Haobo Xu, Yinhe Han, Huawei Li, and Ying Wang. Chipseek: Optimizing verilog generation via eda-integrated reinforcement learning. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2026c.

Tairan Cheng, Yi Liu, Dongsheng Zuo, Zhengyuan Shi, Hongji Zhang, Xiangfei Hu, Maoshuo He, Hao Yan, and Qiang Xu. Hint: Toward an executable hardware-intent representation layer for llm-driven rtl generation. *arXiv preprint arXiv:2608.07625*, 2026.

cocotb contributors. cocotb: Python-based chip (RTL) verification, 2025. URL https://github.com/cocotb/cocotb.

Angela Cui, Ferran Hermida-Rivera, Jack Toubes, Raghav Gupta, Jim Fang, Chengyi Lux Zhang, Ella Schwarz, Junha Kim, Yakun Sophia Shao, Borivoje Nikolić, Christopher W. Fletcher, and Sagar Karandikar. Chia: An open-source framework for principled, agentic ai-driven hardware/software co-design research. *arXiv preprint arXiv:2606.27350*, 2026.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Demin Song, Dahua Lin, Xingcheng Zhang, and Yun (Eric) Liang. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. In *Proceedings of the International Conference on Computer-Aided Design*, 2024.

Stef Cuyckens, Mihaela Jivanescu, Jun Yin, Chao Fang, and Marian Verhelst. Ares: Adaptive reasoning-effort steering for ppa- and cost-aware rtl optimization with llm agents. *arXiv preprint arXiv:2607.27879*, 2026.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

Chenhui Deng, Yun-Da Tsai, Guan-Ting Liu, Zhongzhi Yu, and Haoxing Ren. Scalertl: Scaling llms with reasoning data and test-time compute for accurate rtl code generation. In *Proceedings of the ACM/IEEE Symposium on Machine Learning for CAD*, 2025.

Chenhui Deng, Zhongzhi Yu, Guan-Ting Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Ace-rtl: When agentic context evolution meets rtl-specialized llms. *arXiv preprint arXiv:2602.10218*, 2026.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, Jiajie Jin, Yutao Zhu, Hanbin Wang, Fangyu Lei, Qinyu Luo, Mingyang Chen, Zehui Chen, Jiazhan Feng, Ji-Rong Wen, and Zhicheng Dou. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

Wenji Fang, Yao Lu, Shang Liu, Jing Wang, Ziyan Guo, Junxian He, Fengbin Tu, and Zhiyao Xie. Dr. rtl: Autonomous agentic rtl optimization through tool-grounded self-improvement. *arXiv preprint arXiv:2604.14989*, 2026.

Lik Tung Fu, Jie Zhou, Shaokai Ren, Mengli Zhang, Jia Xiong, Hugo Jiang, Nan Guan, Xi Wang, and Jun Yang. Chatsva: Bridging sva generation for hardware verification via task-specific llms. *arXiv preprint arXiv:2604.02811*, 2026.

Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. AutoVCoder: A systematic framework for automated Verilog code generation using LLMs. In *Proceedings of the IEEE International Conference on Computer Design*, 2024.

Razine Moundir Ghorab, Emanuele Parisi, Cristian Gutierrez, Miquel Alberti-Binimelis, Miquel Moreto, Dario Garcia-Gasulla, and Gokcen Kestor. Notsotiny: A large, living benchmark for rtl code generation. *arXiv preprint arXiv:2512.20823*, 2025.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

Siqi Guo, Ming Lin, and Tianbao Yang. Drtriton: Large-scale synthetic data driven reinforcement learning for triton kernel generation. *arXiv preprint arXiv:2603.21465*, 2026.

Raghu Vamshi Hemadri, Jitendra Bhandari, Andre Nakkab, Johann Knechtel, Badri P Gopalan, Ramesh Narayanaswamy, Ramesh Karri, and Siddharth Garg. Veriloc: Line-of-code level prediction of hardware design quality from verilog code. In *Advances in Neural Information Processing Systems*, 2025.

Wei-Po Hsin, Ren-Hao Deng, Yao-Ting Hsieh, En-Ming Huang, and Shih-Hao Hung. Evolve: Evolutionary search for llm-based verilog generation and optimization. *arXiv preprint arXiv:2601.18067*, 2026.

Yuchen Hu, Junhao Ye, Ke Xu, Jialin Sun, Shiyue Zhang, Xinyao Jiao, Dingrong Pan, Jie Zhou, Ning Wang, Weiwei Shan, Xinwei Fang, Xi Wang, Nan Guan, and Zhe Jiang. Uvllm: An automated universal rtl verification framework using llms. In *Proceedings of the Design Automation Conference*, 2025.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. In *Proceedings of the International Conference on Learning Representations*, 2026.

IEEE. IEEE standard for universal verification methodology language reference manual. IEEE Std 1800.2-2020, IEEE, 2020.

Sazzadul Islam, Tasnim Tabassum, and Hao Zheng. Verigraphi: A multi-agent framework of hierarchical rtl generation for large hardware designs. *arXiv preprint arXiv:2604.14550*, 2026.

Pengwei Jin, Di Huang, Chongxiao Li, Shuyao Cheng, Yang Zhao, Xinyao Zheng, Jiaguo Zhu, Shuyi Xing, Bohan Dou, Rui Zhang, Zidong Du, Qi Guo, and Xing Hu. Realbench: Benchmarking verilog generation models with real-world ip designs. *arXiv preprint arXiv:2507.16200*, 2025.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Transactions on Machine Learning Research*, 2023.

Kezhi Li, Min Li, Xiangyu Wen, Shibo Zhao, Jieying Wu, Junhua Huang, and Qiang Xu. Formalrtl: Verified rtl synthesis at scale. *arXiv preprint arXiv:2603.08738*, 2026.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *Proceedings of the International Conference on Computer-Aided Design*, 2023.

Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, Bonita Bhaskaran, Bryan Catanzaro, Arjun Chaudhuri, Sharon Clay, Bill Dally, Laura Dang, Parikshit Deshpande, Siddhanth Dhodhi, Sameer Halepete, Eric Hill, Jiashang Hu, Sumit Jain, Ankit Jindal, Brucek Khailany, George Kokai, Kishor Kunal, Xiaowei Li, Charley Lind, Hao Liu, Stuart Oberman, Sujeet Omar, Ghasem Pasandi, Sreedhar Pratty, Jonathan Raiman, Ambar Sarkar, Zhengjiang Shao, Hanfei Sun, Pratik P Suthar, Varun Tej, Walker Turner, Kaizhe Xu, and Haoxing Ren. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176*, 2024a.

Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. In *Proceedings of the International Conference on Learning Representations*, 2025.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. In *Proceedings of the IEEE LLM Aided Design Workshop*, 2024b.

Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation. In *Proceedings of the International Conference on Computer-Aided Design*, 2024c.

Hongliang Lu, Yuhang Wen, Pengyu Cheng, Ruijin Ding, Jiaqi Guo, Haotian Xu, Chutian Wang, Haonan Chen, Xiaoxi Jiang, and Guanjun Jiang. Search self-play: Pushing the frontier of agent capability without supervision. In *Proceedings of the International Conference on Learning Representations*, 2026a.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 2024.

Yao Lu, Shang Liu, Hangan Zhou, Wenji Fang, Qijun Zhang, and Zhiyao Xie. A new benchmark for the appropriate evaluation of rtl code optimization. *arXiv preprint arXiv:2601.01765*, 2026b.

Ruiyang Ma, Yuxin Yang, Ziqian Liu, Jiaxi Zhang, Min Li, Junhua Huang, and Guojie Luo. Verilogreader: Llm-aided hardware test generation. In *Proceedings of the IEEE LLM Aided Design Workshop*, 2024.

Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the International Conference on Software Engineering*, 2007.

Bhabesh Mali, Karthik Maddala, Vatsal Gupta, Sweeya Reddy, Chandan Karfa, and Ramesh Karri. Chiraag: Chatgpt informed rapid and automated assertion generation. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2024.

Jianan Mu, Mingyu Shi, Yining Wang, Tianmeng Yang, Bin Sun, Xing Hu, Jing Ye, and Huawei Li. Faver: Boosting llm-based rtl generation with function abstracted verifiable middleware. *arXiv preprint arXiv:2510.08664*, 2025.

Nangate. Nangate 45nm open cell library, 2008. URL `https://si2.org/open-cell-and-free-pdk-libraries/`.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI Magazine*, 2007.

Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. Using llms to facilitate formal verification of rtl. *arXiv preprint arXiv:2309.09437*, 2023.

Parallax Software, Inc. OpenSTA: Static timing analyzer, 2026. URL `https://github.com/parallaxsw/OpenSTA`.

Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. Betterv: Controlled verilog generation with discriminative guidance. In *Proceedings of the International Conference on Machine Learning*, 2024.

Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting verilogeval: A year of improvements in large-language models for hardware code generation. *ACM Transactions on Design Automation of Electronic Systems*, 2025a.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on rtl design and verification. *arXiv preprint arXiv:2506.14074*, 2025b.

Heng Ping, Peiyu Zhang, Zhenkun Wang, Shixuan Li, Anzhe Cheng, Wei Yang, Paul Bogdan, and Shahin Nazarian. Poet: Power-oriented evolutionary tuning for llm-based rtl ppa optimization. *arXiv preprint arXiv:2603.19333*, 2026.

Suresh Purini, Siddhant Garg, Mudit Gaur, Sankalp Bhat, Sohan Mupparapu, and Arun Ravindran. Archxbench: A complex digital systems benchmark suite for llm driven rtl synthesis. In *Proceedings of the ACM/IEEE Symposium on Machine Learning for CAD*, 2025.

Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. Autobench: Automatic testbench generation and evaluation using llms for hdl design. In *Proceedings of the ACM/IEEE International Symposium on Machine Learning for CAD*, 2024.

Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. Correctbench: Automatic testbench generation with functional self-correction using llms for hdl design. In *Proceedings of the Design, Automation and Test in Europe Conference*, 2025.

Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Tsungyi Ho, Ulf Schlichtmann, and Bing Li. Confibench: Automatic testbench generation with confidence-based scenario mask and testbench ensemble using llms for hdl design. *ACM Transactions on Design Automation of Electronic Systems*, 2026.

Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolić. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2nd edition, 2003.

Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the European Software Engineering Conference Held Jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Jiahe Shi, Zhengqi Gao, Ching-Yun Ko, and Duane Boning. Earl: Entropy-aware rl alignment of llms for reliable rtl code generation. *arXiv preprint arXiv:2511.12033*, 2025.

Zhengyuan Shi, Jingxin Wang, Tairan Cheng, Changran Xu, Weikang Qian, and Qiang Xu. Cktevo: Repository-level rtl code benchmark for design evolution. *arXiv preprint arXiv:2603.08718*, 2026.

SkyWater PDK Authors. SkyWater open source PDK, 2020. URL `https://github.com/google/skywater-pdk`.

Wilson Snyder, Paul Wasson, Duane Galbi, Geza Lore, et al. Verilator, 2026. URL `https://github.com/verilator/verilator`.

James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. FreePDK: An open-source variation-aware design kit. In *Proceedings of the IEEE International Conference on Microelectronic Systems Education*, 2007.

Yan Tan, Xiangchen Meng, Zijun Jiang, and Yangdi Lyu. Autoverifix: Automatically correcting errors and enhancing functional correctness in llm-generated verilog code. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 2026a.

Yan Tan, Xiangchen Meng, Zijun Jiang, and Yangdi Lyu. Autoverifix+: High-correctness rtl generation via trace-aware causal fix and semantic redundancy pruning. *arXiv preprint arXiv:2603.11489*, 2026b.

Kimia Tasnia, Alexander Garcia, Tasnuva Farheen, and Sazadur Rahman. Veriopt: Ppa-aware high-quality verilog generation via multi-role llms. In *Proceedings of the International Conference on Computer-Aided Design*, 2025.

Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *Proceedings of the Design, Automation and Test in Europe Conference*, 2023.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 2024.

Kiran Thorat, Jiahui Zhao, Yaotian Liu, Amit Hasan, Hongwu Peng, Xi Xie, Bin Lei, and Caiwen Ding. Llm-verippa: Power, performance, and area optimization aware verilog code generation with large language models. In *Proceedings of the ACM/IEEE Symposium on Machine Learning for CAD*, 2025.

Yunda Tsai, Mingjie Liu, and Haoxing Ren. Rtlfixer: Automatically fixing rtl syntax errors with large language model. In *Proceedings of the Design Automation Conference*, 2024.

Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Nan Guan, and Zhe Jiang. Insights from verification: Training a verilog generation llm with reinforcement learning with testbench feedback. *arXiv preprint arXiv:2504.15804*, 2025a.

Ning Wang, Bingkun Yao, Jie Zhou, Yuchen Hu, Xi Wang, Zhe Jiang, and Nan Guan. Large language model for verilog generation with code-structure-guided reinforcement learning. In *Proceedings of the IEEE International Conference on LLM-Aided Design*, 2025b.

Shaobo Wang, Zhengbo Jiao, Zifan Zhang, Yilang Peng, Xu Ze, Boyu Yang, Wei Wang, Hu Wei, and Linfeng Zhang. Socratic-zero : Bootstrapping reasoning via data-free agent co-evolution. *arXiv preprint arXiv:2509.24726*, 2025c.

Yaoxiang Wang, Qi Shi, Shangzhan Li, Qingguo Hu, Xinyu Yin, Bo Guo, Xu Han, Maosong Sun, and Jinsong Su. Veriagent: A tool-integrated multi-agent system with evolving memory for ppa-aware rtl code generation. *arXiv preprint arXiv:2603.17613*, 2026a.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. In *Advances in Neural Information Processing Systems*, 2025d.

Yiting Wang, Wanghao Ye, Ping Guo, Yexiao He, Ziyao Wang, Bowei Tian, Shwai He, Guoheng Sun, Zheyu Shen, Sihan Chen, Ankur Srivastava, Qingfu Zhang, Gang Qu, and Ang Li. Symrtlo: Enhancing rtl code optimization with llms and neuron-inspired symbolic reasoning. In *Advances in Neural Information Processing Systems*, 2025e.

Yiting Wang, Wanghao Ye, Yexiao He, Yiran Chen, Gang Qu, and Ang Li. Mcp4eda: Llm-powered model context protocol rtl-to-gdsii automation with backend aware synthesis optimization. *arXiv preprint arXiv:2507.19570*, 2025f.

Yiting Wang, Guoheng Sun, Wanghao Ye, Gang Qu, and Ang Li. Verireason: Reinforcement learning with testbench feedback for reasoning-enhanced verilog generation. In *Proceedings of the Great Lakes Symposium on VLSI*, 2026b.

Anjiang Wei, Huanmi Tan, Tarun Suresh, Daniel Mendoza, Thiago S. F. X. Teixeira, Ke Wang, Caroline Trippel, and Alex Aiken. Vericoder: Enhancing llm-based rtl code generation through functional correctness validation. In *Workshop on Deep Learning for Code at NeurIPS*, 2025a.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems*, 2025b.

Bruce Wile, John Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, 2005.

Stephen Williams. Icarus Verilog, 2026. URL `https://github.com/steveicarus/iverilog`.

Clifford Wolf and Johann Glaser. Yosys - a free verilog synthesis suite. In *Proceedings of the Austrian Workshop on Microelectronics*, 2013.

Yuheng Wu, Berk Gokmen, Zhouhua Xie, Peijing Li, Caroline Trippel, Priyanka Raina, and Thierry Tambe. Llm-fsm: Scaling large language models for finite-state reasoning in rtl code generation. *arXiv preprint arXiv:2602.07032*, 2026a.

Yutong Wu, Chenrui Cao, Pengwei Jin, Di Huang, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, and Xing Hu. Qimeng-codev-sva: Training specialized llms for hardware assertion generation via rtl-grounded bidirectional data synthesis. *arXiv preprint arXiv:2603.14239*, 2026b.

Xin Xin, Jincheng Lou, Junhui Li, Jinglin Yan, Panda Xiao, Di Wu, Haixiao Li, Weicong Lu, Weijian Fan, Xinyu Qu, Yuxiang Zhao, Min Yu, Zhixiong Di, and Yibo Lin. Gogotb: Agentic rtl verification with specification-grounded coverage closure. *arXiv preprint arXiv:2607.26181*, 2026.

Kangwei Xu, Grace Li Zhang, Ulf Schlichtmann, and Bing Li. Correcthdl: Agentic hdl design with llms leveraging high-level synthesis as reference. *arXiv preprint arXiv:2511.16395*, 2025.

Peiran Yan, Qinzhe Zhi, Lifeng Liu, and Tianyu Jia. Gensoc: A multi-agent-assisted soc generation methodology leveraging open-source hardware. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*, 2025a.

Zhiling Yan, Dingjie Song, Hanrong Zhang, Wei Liang, Yuxuan Zhang, Yutong Dai, Lifang He, Philip S. Yu, Ran Xu, Xiang Li, and Lichao Sun. Openskill: Open-world self-evolution for llm agents. *arXiv preprint arXiv:2606.06741*, 2026.

Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. Assertllm: Generating hardware verification assertions from design specifications via multi-llms. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 2025b.

Chengyi Yang, Zhishang Xiang, Yunbo Tang, Zongpei Teng, Chengsong Huang, Fei Long, Yuhan Liu, and Jinsong Su. Ttcs: Test-time curriculum synthesis for self-evolving. *arXiv preprint arXiv:2601.22628*, 2026.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

Xufeng Yao, Haoyang Li, Tsz Ho Chan, Wenyi Xiao, Mingxuan Yuan, Yu Huang, Lei Chen, and Bei Yu. Hdldebugger: Streamlining hdl debugging with large language models. *ACM Transactions on Design Automation of Electronic Systems*, 2025.

Jianmin Ye, Tianyang Liu, Qi Tian, Shengchu Su, Zhe Jiang, and Xi Wang. Chatmodel: Automating reference model design and verification with llms. *arXiv preprint arXiv:2506.15066*, 2025a.

Junhao Ye, Yuchen Hu, Ke Xu, Dingrong Pan, Qichun Chen, Jie Zhou, Shuai Zhao, Xinwei Fang, Xi Wang, Nan Guan, and Zhe Jiang. From concept to practice: an automated llm-aided uvm machine for rtl verification. In *Proceedings of the International Conference on Computer-Aided Design*, 2025b.

Junhao Ye, Dingrong Pan, Hanyuan Liu, Yuchen Hu, Jie Zhou, Ke Xu, Xinwei Fang, Xi Wang, Nan Guan, and Zhe Jiang. Uvmarvel: an automated llm-aided uvm machine for subsystem-level rtl verification. *arXiv preprint arXiv:2605.04704*, 2026.

Chenyang Yin, Agasthi Haputhanthri, Aditya Anirudh Jonnalagadda, Zhenyu Bai, Yuanming Song, Saranyu Chattopadhyay, Mohammad Fadiheh, Tom Zelazny, Subhasish Mitra, and Tulika Mitra. Llm-based hardware development with hierarchical irs and end-to-end multi-agent workflow. *arXiv preprint arXiv:2608.30659*, 2026.

Cunxi Yu, Chenhui Deng, Nathaniel Pinckney, and Brucek Khailany. Agentic hardware design as repository-level code evolution. *arXiv preprint arXiv:2606.28279*, 2026a.

Zhongkai Yu, Yichen Lin, Chenyang Zhou, Yuwei Zhang, Kun Zhou, Junxia Cui, Haotian Ye, Zhengding Hu, Zaifeng Pan, Ruiyi Wang, Yujie Zhao, Hejia Zhang, Jingbo Shang, Jishen Zhao, and Yufei Ding. Chipmate: Multi-agent training via reinforcement learning for enhanced rtl generation. *arXiv preprint arXiv:2605.12857*, 2026b.

Zhongkai Yu, Chenyang Zhou, Yichen Lin, Hejia Zhang, Haotian Ye, Junxia Cui, Zaifeng Pan, Jishen Zhao, and Yufei Ding. Chipbench: A next-step benchmark for evaluating llm performance in ai-aided chip design. *arXiv preprint arXiv:2601.21448*, 2026c.

Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine Lin, Yong Liu, and Haoxing Ren. Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems. In *Proceedings of the IEEE International Conference on LLM-Aided Design*, 2025.

Patrick Yubeaton, Andre Nakkab, Weihua Xiao, Luca Collini, Ramesh Karri, Chinmay Hegde, and Siddharth Garg. Verithoughts: Enabling automated verilog code generation using reasoning and formal verification. In *Advances in Neural Information Processing Systems*, 2025.

Patrick Yubeaton, Siddharth Garg, and Chinmay Hegde. Exploring the agentic frontier of verilog code generation. *arXiv preprint arXiv:2603.19347*, 2026.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. Accelopt: A self-improving llm agentic system for ai accelerator kernel optimization. In *Proceedings of Machine Learning and Systems*, 2026a.

Hejia Zhang, Zhongming Yu, Chia-Tung Ho, Haoxing Ren, Brucek Khailany, and Jishen Zhao. Llm4cov: Execution-aware agentic learning for high-coverage testbench generation. In *Proceedings of the International Conference on Machine Learning*, 2026b.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *Proceedings of the International Conference on Learning Representations*, 2026c.

Lunjun Zhang, Ryan Chen, and Bradly C. Stadie. Evolutionary system prompt learning for reinforcement learning in llms. *arXiv preprint arXiv:2602.14697*, 2026d.

Xinyu Zhang, Zhiteng Chao, Yonghao Wang, Bin Sun, Tianyun Ma, Tianmeng Yang, Jianan Mu, Jing Justin Ye, and Huawei Li. Rtlseek: Boosting the llm-based rtl generation with multi-stage diversity-oriented reinforcement learning. *arXiv preprint arXiv:2603.27630*, 2026e.

Yang Zhang, Rui Zhang, Jiaming Guo, Lei Huang, Di Huang, Yunpu Zhao, Shuyao Cheng, Pengwei Jin, Chongxiao Li, Zidong Du, Xing Hu, Qi Guo, and Yunji Chen. Qimeng-salv: Signal-aware learning for verilog code generation. In *Advances in Neural Information Processing Systems*, 2025a.

Yifan Zhang, Jianmin Ye, Jiahao Yang, and Xi Wang. Refevo: Agentic design with co-evolutionary verification for agile reference model generation. *arXiv preprint arXiv:2604.24218*, 2026f.

Zixi Zhang, Balint Szekely, Pedro Gimenes, Greg Chadwick, Hugo McNally, Jianyi Cheng, Robert Mullins, and Yiren Zhao. Llm4dv: Using large language models for hardware test stimuli generation. In *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 2025b.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems*, 2025a.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, and Xing Hu. Codev: Empowering llms with hdl generation through multilevel summarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2026a.

Yujie Zhao, Zhijing Wu, Boqin Yuan, Zhongming Yu, Hejia Zhang, Wentao Ni, Chia-Tung Ho, Haoxing Ren, and Jishen Zhao. Pro-v-r1: Reasoning enhanced programming agent for rtl verification. *arXiv preprint arXiv:2506.12200*, 2025b.

Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. Mage: A multi-agent engine for automated rtl code generation. In *Proceedings of the Design Automation Conference*, 2025c.

Zhe Zhao, Hongbing Lang, Zhihan Xiao, Luke Ztz Hu, John Imoleayo Adebisi, and Songping Mai. Evidence-driven llm agent for c-to-synthesizable-c conversion and verification. *arXiv preprint arXiv:2606.28409*, 2026b.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, 2024.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, Pengwei Jin, Shuyao Cheng, Shengwen Liang, Xishan Zhang, Rui Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. Qimeng-codev-r1: Reasoning-enhanced verilog generation. In *Advances in Neural Information Processing Systems*, 2025a.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling, 2025b. URL `https://github.com/THUDM/slime`.

APPENDIX CONTENTS

Appendix A: BEHAVE: System Design 20

A.1 End-to-End Workflow 20
A.2 Task and Artifact Example 20
A.3 Execution Architecture 21
A.4 Supported Features and Limitations 21
A.5 Transaction Mapping and Output Matching 24
A.6 Goals, Coverage and Scoring 26
A.7 Goal-Guided Exploration and Evaluation 28

Appendix B: BEHAVE-Sim Formal Semantics and Bounded Guarantees 31

B.1 Trace Domains and Goal Reachability 31
B.2 Finite Representations and Exact Encodings 33
B.3 Bounded Reachability Analysis 35
B.4 RTL Event-Slot Semantics and Protocol Adapters 38
B.5 RTL Closed-Loop Semantics and Trace Domain 40
B.6 RTL Finite Representation, Exact Encoding and Audits 40
B.7 Behavior IR and Procedure Semantics 42
B.8 Behavior Finite Representation and Exact Encoding 44
B.9 Behavior Frontend Fidelity and Audits 49
B.10 Artifact Instances and Reported Bounded Conclusions 50
B.11 Trusted Boundary, Claim Scope and Conclusion 51

Appendix C: Implementation, Experimental Details and Additional Results 52

C.1 Agent Environment and Tools 52
C.2 Inference and Evaluation Settings 52
C.3 Effect of Goal-Guided Search on RTL Coverage 53
C.4 Behavior IR and HLS Comparison Settings 53
C.5 Pipeline Comparison and PPA Exploration Settings 53
C.6 Training and Self-Improvement Settings 54
C.7 Computational Resources and Cost 55
C.8 Pipeline Comparison Example 55
C.9 Additional Results for RL Training and Self-Improvement 56

Appendix D: Dataset Construction and Validation 56

D.1 Dataset Overview and Sources 56
D.2 Source-Based Task Construction 58
D.3 External Benchmark Adaptation 59
D.4 Human Review 60
D.5 Data Splits and Release 60

Appendix E: Extended Related Work 61

A BEHAVE: SYSTEM DESIGN

Purpose. We describe how BEHAVE supports joint RTL and behavior development and independently evaluates artifacts against the golden behavior model under task-defined protocol requirements.

Overview. Subsections A.1 through A.4 describe the workflow, an artifact example, execution architecture and supported features. Subsections A.5 through A.7 define transaction mapping, goals, scoring and evaluation, from baseline execution to goal-guided search and continuous workloads.

A.1 END-TO-END WORKFLOW

Development and evaluation. Figure 8 connects joint development to independent evaluation. Given S, the agent develops $\hat{R}$, $\hat{B}$, and a testbench through tool interaction. After submission, BEHAVE-Sim uses the task interface and hidden $B^\star$ to evaluate $\hat{R}$ and $\hat{B}$ separately. Stimulus generation and comparison follow Figure 2(c) and Appendix A.7, which describes baseline-first search and continuous workloads. We separately report the testbench's coverage of the agent's own RTL.

Training and self-improvement. Training rollouts supply reward r for RL updates. Development-set feedback guides acquisition of new $(S, B^\star)$ pairs for further training, and the updated agent guides subsequent acquisition rounds. The held-out set remains fixed for evaluation. Optional bounded analysis is separate from scoring and rewards, as described in Appendix B.

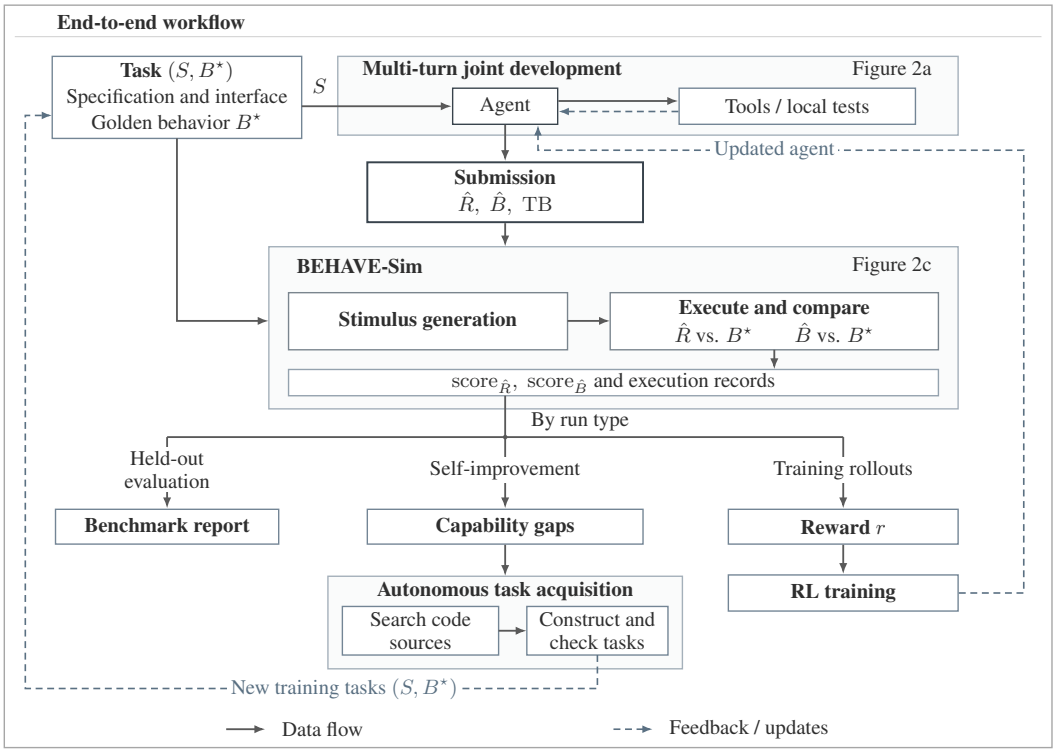


Figure 8: **End-to-end workflow.** The shared evaluator supports held-out reporting, capability diagnosis for self-improvement, and reward computation for training rollouts.

A.2 TASK AND ARTIFACT EXAMPLE

Task. The MXFP4 block-quantization task accepts 32 BF16 encodings and returns a shared E8M0 scale and 32 E2M1 codes packed into 16 bytes. The specification defines scale selection, round-to-nearest-even quantization, saturation and packing order. Its non-finite-input policy assigns scale code 255 and zero data to any block containing NaN or infinity.

```python
class ai_ml_quant_mx_mxfp4_block_quant:
    def __init__(self):
        self.reset()

    def reset(self):
        pass

    def process(self, in_x0, ..., in_x31):
        xs = [in_x0, ..., in_x31]
        maximum = 0
        for i in range(32):
            magnitude = xs[i] & 32767
            if magnitude > maximum:
                maximum = magnitude
        scale = (maximum >> 7) - 2
        if scale < 0:
            scale = 0
        if maximum >= 32640:
            scale = 255
        q = [0] * 32
        for i in range(32):
            q[i] = quantize_one(xs[i], scale)
        return {
            "out_scale": scale,
            "out_q0": q[0] | (q[1] << 4),
            # Remaining output fields follow the same packing rule.
            "out_q15": q[30] | (q[31] << 4)
        }
```

Figure 9: **Behavior model structure and computation.** Ellipses abbreviate repeated interface fields. The bit-exact element quantizer `quantize_one` is omitted.

Behavior and RTL. Figures 9 and 10 show the behavior model and one RTL implementation. The model selects the shared scale, quantizes the inputs and packs the outputs in `process()`, without retaining state across calls. The RTL registers each block and its scale before quantization and output packing, forming a two-stage pipeline. Both stages stall under output backpressure.

Interface alignment. Figure 10 shows an RTL implementation. Other designs may use different pipeline depths or resource sharing while respecting the task's protocol and timing constraints. The same behavior model can check these designs through protocol-defined input mapping and output matching.

A.3 Execution Architecture

RTL execution. As shown in Figure 11, BEHAVE-Sim executes $\hat{R}$ with an input driver, an output receiver, clock/reset control, and a task-declared external-memory model when needed. The stimulus supplies input transactions and schedules their presentation, output readiness, and clock/reset events. Passive interface observations feed task-defined input/output mapping and protocol/timing monitors. BEHAVE-Sim records logical I/O and collects goal hits from monitors and RTL instrumentation.

Behavior execution. For $\hat{B}$, logical inputs and task-visible events directly drive model invocations. State persists between invocations, with resets applied as declared by the task. BEHAVE-Sim records outputs and goal hits, while $B^{\star}$ runs with independent state on the same input history to compute expected outputs. Behavior and RTL evaluation use the same output-matching and scoring rules.

A.4 Supported Features and Limitations

Interfaces and protocols. Table 3 summarizes interface capabilities and representative protocols. Each task specifies its protocol configurations and may combine capabilities across its interfaces.

```
logic valid_scale, advance;
logic [15:0] xs [0:31], xs_s1 [0:31];
logic [14:0] maximum;
logic [7:0] scale, scale_s1;
logic [3:0] q [0:31];

assign advance = !out_valid || out_ready;
assign in_ready = advance;

always_comb begin
    xs[0] = in_x0;
    // The remaining input lanes are wired identically.
    xs[31] = in_x31;

    maximum = 0;
    for (integer i = 0; i < 32; i = i + 1)
        if (xs[i][14:0] > maximum)
            maximum = xs[i][14:0];
    scale = maximum[14:7] < 2 ? 0 : maximum[14:7] - 8'd2;
    if (maximum >= 15'h7F80) scale = 255;
    for (integer i = 0; i < 32; i = i + 1)
        q[i] = quantize(xs_s1[i], scale_s1);
end

always_ff @(posedge clk or posedge rst) begin
    if (rst) begin
        valid_scale <= 0;
        out_valid <= 0;
        scale_s1 <= 0;
        for (integer i = 0; i < 32; i = i + 1) xs_s1[i] <= 0;
        out_scale <= 0;
        out_q0 <= 0;
        // out_q1 through out_q14 are also cleared.
        out_q15 <= 0;
    end else if (advance) begin
        valid_scale <= in_valid;
        out_valid <= valid_scale;
        if (in_valid) begin
            scale_s1 <= scale;
            for (integer i = 0; i < 32; i = i + 1)
                xs_s1[i] <= xs[i];
        end
        if (valid_scale) begin
            out_scale <= scale_s1;
            out_q0 <= {q[1], q[0]};
            // out_q1 through out_q14 pack the remaining pairs.
            out_q15 <= {q[31], q[30]};
        end
    end
end
```

Figure 10: **Two-stage pipelined RTL.** Port declarations, repeated lane assignments and the bit-exact element quantizer `quantize` are omitted.

Transaction mapping. The task defines how RTL interface events form logical inputs to the behavior model and how observed outputs are matched with its predictions. A logical operation may involve one transfer or several transfers. This mapping is fixed before evaluation.

Interface composition. A task may combine different interfaces, such as configuration and data interfaces. It specifies when configuration changes take effect and how simultaneous interactions are handled, so that the behavior model captures their combined effect.

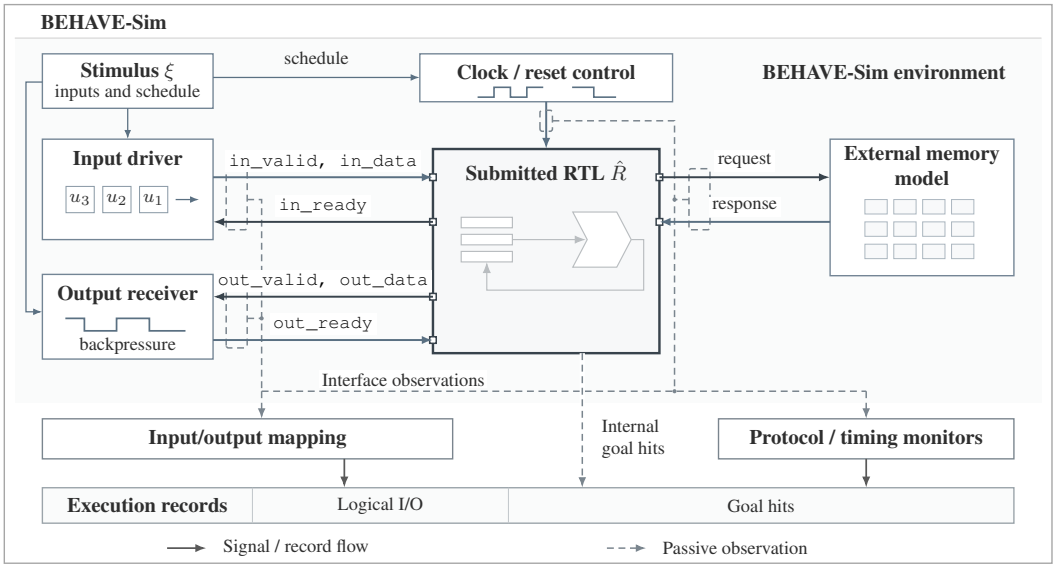


Figure 11: **RTL execution environment in BEHAVE-Sim.** Expanded view of the RTL execution in Figure 2(c), using ready/valid interfaces and an optional task-declared external-memory model with a schematic request/response interface. Task-defined mapping and monitors record logical I/O, task-visible events, and goal hits. Internal goal hits are collected by instrumentation.

Table 3: **Interface capabilities and representative protocols.**

Capability	Interaction requirements	Typical interfaces
Sampling and flow control	Sampling events, transfer detection and backpressure	Plain, valid-only and ready/valid interfaces
Framing and byte validity	Frame boundaries, valid bytes and sideband fields	AXI4-Stream (Arm, 2010)
Phased transfers	Setup, access, wait states and completion	APB (Arm, 2023)
Multi-channel assembly	Independently arriving address and data	AXI4-Lite (Arm, 2020)
Outstanding requests	Pending-request limits and response ordering	Memory request/response interfaces
Invocation and completion	Start, busy, completion and result consumption	Start/done interfaces
Multi-interface composition	Mixed protocols and cross-interface ordering	AXI4-Lite configuration with AXI4-Stream data

Capability separation. Table 4 distinguishes the features supported by evaluation and replay, goal-guided stimulus generation and exact bounded analysis. Generation and exact analysis require their own encodings of the selected protocol and transaction mapping. Goal-guided generation may extend the evaluation stimulus pool, whose concrete execution records determine the artifact score. Exact bounded analysis is optional, runs after scoring and changes neither the evaluation pool nor the score.

Unsupported and unresolved cases. If a symbolic capability cannot represent an artifact, goal or schedule, it refuses without affecting evaluation and replay. A generation query finding no candidate adds no stimulus and proves nothing about reachability. A timeout or unknown result in exact analysis leaves the corresponding goal unresolved. None of these outcomes changes a completed artifact score. An artifact failure during concrete evaluation replay remains a scored evaluation failure.

Table 4: **BEHAVE-Sim features and limitations.** Rows list artifact features, and the remaining columns report support for evaluation and replay, goal-guided generation and exact bounded analysis.

Feature	Evaluation and replay	Goal-guided generation	Exact bounded analysis
Behavior models	Behavior models use a restricted Python subset. I/O values may be Boolean, bounded integers, fixed-size arrays, records or enumerations.	The `process()` implementation must have statically proved finite bounds.	Every analyzed operation and loop must have a finite exact encoding.
RTL designs	Fixed-width SystemVerilog designs that pass structural checks execute in isolated Verilator. Native source-event observation does not require a statically proved loop trip count.	Each target goal requires a finite symbolic query relating selected input fields to whether that goal is reached.	A faithful encoding must cover the complete RTL transition relation within the analysis bound.
Protocols	Behavior uses logical transactions. RTL supports plain, valid-only, ready/valid and start/done interfaces, task-configured APB, AXI4-Lite and framed AXI4-Stream subsets, and bounded request/response.	Requires a finite symbolic query covering the selected protocol and transaction mapping. Generation varies logical input fields under fixed protocol rules, transaction order and environment schedule.	Requires an exact encoding of the selected protocol, transaction mapping and monitors. Schedules remain fixed except for single-clock ready/valid with one input and one output.
Endpoints and clock domains	Replay supports multiple endpoints and clock domains for Behavior and RTL. A task's RTL interfaces may use different protocols.	Behavior generation supports multiple endpoints within one clock domain. RTL generation with multiple endpoints or clocks requires a fixed global schedule.	Behavior analysis supports multiple endpoints. Multi-clock Behavior and multiple-endpoint or multi-clock RTL require all analyzed stimuli to share one complete global clock schedule.
Reset	Replay follows the reset events in each stimulus. Reset during traffic requires the task to define how pending transactions are handled.	Goal-guided generation does not add or move reset events.	Reset timing remains fixed unless it is explicitly represented as a symbolic schedule choice.
State and internal memory	Replay supports persistent Behavior state, RTL registers and writable internal memories.	Behavior state must be finite and initialized. RTL goal-guided generation does not support writable internal memories.	All analyzed state must have a finite exact encoding. RTL exact analysis does not support writable internal memories.
External memory	Replay supports a task-defined external-memory model for RTL and single-domain Behavior models.	Goal-guided generation is unavailable for artifacts that access external memory.	Exact analysis supports external memory only for RTL, using a fixed finite memory model.
Goals and monitors	Replay records reached coverage goals and detected safety violations. For RTL, monitor violations are recorded as safety-goal hits.	A goal is targeted only when a finite symbolic query can propose inputs that reach it.	Every catalogued goal, including monitor violations, must have an exact reachability encoding.

A.5 Transaction Mapping and Output Matching

Overview. BEHAVE-Sim evaluates $\hat{B}$ and $\hat{R}$ separately against $B^\star$ on stimulus sets. For each stimulus, it records which inputs the target accepts under each schedule entry, obtains the expected output histories from $B^\star$, and matches them against the target's observed output histories. This subsection formalizes stimuli, accepted inputs, output histories, and matching.

Definition 1 (Transaction interface)**.** An *endpoint* c of an artifact A is a named input or output with a nonempty finite type T_c. A *transaction* on c is a value $x \in T_c$ exchanged between A and its environment. Transactions on each endpoint form an ordered stream. The task declares finite families of input endpoints C^{in} and output endpoints C^{out}, which form the logical transaction interface.

Definition 2 (Stimulus)**.** For each input endpoint c, let

$$\mathbf{i}_c = (x_{c,0}, \ldots, x_{c,k_c-1}) \in T_c^{k_c}$$

be an ordered queue of k_c transactions, and write $\mathbf{i} = (\mathbf{i}_c)_{c \in C^{\text{in}}}$ for the queue family. A *stimulus* is a pair $\xi = (\mathbf{i}, \sigma)$, where

$$\sigma = (\sigma_t)_{0 \le t < m}$$

is a finite schedule of length m. Each entry σ_t specifies clock and reset events and an environment action. The schedule controls queue presentation, allowing simultaneous presentation across endpoints while preserving each queue's order. A runtime request supplies $\mathbf{i}$ and an execution policy, recording σ during execution. Pending inputs are offered until accepted. After all inputs are accepted and expected outputs collected, a finite observation tail follows. Task-declared timing requirements are checked separately. Resource exhaustion alone is not a task violation.

Definition 3 (Accepted input trace)**.** Fix $A \in \{\hat{B}, \hat{R}\}$ and a stimulus $\xi = (\mathbf{i}, \sigma)$ with $|\sigma| = m$. The index t identifies a schedule entry, not a cycle count for any particular clock domain. For $0 \le t < m$ and $c \in C^{\text{in}}$, let $\perp$ be a distinguished symbol outside every transaction type and let

$$a^A_{t,c}(\xi) \in T_c \cup \{\perp\}$$

be the transaction from $\mathbf{i}_c$ that A accepts on endpoint c under σ_t, or $\perp$ if it accepts none. The *accepted input batch* under σ_t is

$$a^A_t(\xi) := \big(a^A_{t,c}(\xi)\big)_{c \in C^{\text{in}}} \in \prod_{c \in C^{\text{in}}} \big(T_c \cup \{\perp\}\big).$$

The *accepted input trace* is

$$\mathbf{a}_A(\xi) = \big(a^A_0(\xi), \ldots, a^A_{m-1}(\xi)\big).$$

Thus one batch corresponds to one schedule entry. A batch may be all-$\perp$ when no endpoint accepts a transaction, and several non-$\perp$ entries in one batch are accepted together. For $A = \hat{B}$, the Behavior interface has no input handshake, so every presented input batch is accepted and placed at its scheduled entry in $\mathbf{a}_{\hat{B}}(\xi)$. For $A = \hat{R}$, the non-$\perp$ entries of $\mathbf{a}_{\hat{R}}(\xi)$ are the acceptance events reported by its declared protocol adapter.

Definition 4 (Output histories)**.** For each output endpoint c, let $\ell_{A,c}(\xi)$ be the number of output transactions observed on c under ξ, and write

$$o_{A,c}(\xi) = \big(o_{A,c,0}(\xi), \ldots, o_{A,c,\ell_{A,c}(\xi)-1}(\xi)\big) \in T_c^{\ell_{A,c}(\xi)}.$$

Collect these histories as

$$\mathbf{o}_A(\xi) = \big(o_{A,c}(\xi)\big)_{c \in C^{\text{out}}}.$$

Let π be the task-defined mapping from an execution schedule and an accepted-input trace to the input history supplied to $B^\star$. Thus, $\pi(\sigma, \mathbf{a})$ contains the accepted transactions, their declared ordering, and task-visible events such as resets and external-memory responses, while omitting functionally irrelevant schedule details. The corresponding expected histories are

$$\mathbf{o}^\star_A(\xi) = B^\star\big(\pi(\sigma, \mathbf{a}_A(\xi))\big), \qquad o^\star_{A,c}(\xi) = \big(\mathbf{o}^\star_A(\xi)\big)_c.$$

The task requires $B^\star$ to determine every expected history completely and deterministically, including its length and every compared field. Behavior-model histories are observed directly, while RTL histories are reconstructed from the emission events reported by its protocol adapter.

Matching outputs. For each output endpoint c, the observed history $o_{A,c}(\xi)$ is matched against the expected history $o^\star_{A,c}(\xi)$ under a task-declared policy. By default, transactions are paired in order. If the task permits reordered completion, it may declare an identifier that uniquely distinguishes outstanding transactions, and transactions with the same identifier are paired. An endpoint without such an identifier may instead use content-based matching under Assumption 1. Each unpaired expected or observed transaction and each paired transaction with unequal fields contributes one mismatch. For $A \in \{\hat{B}, \hat{R}\}$, let $n^A_{\text{mis}}(\xi)$ be the total number of mismatches across output endpoints.

Assumption 1 (Validity of content-based matching)**.** For every output endpoint compared by content, output order carries no task-defined meaning. An observed history satisfies the task's output requirement exactly when it contains the same multiset of transaction values as the expected history.

Definition 5 (Evaluation counts)**.** For $A \in \{\hat{B}, \hat{R}\}$, let $n^A_{\text{txn}}(\xi)$ denote the number of input transactions that A accepts under ξ:

$$n^A_{\text{txn}}(\xi) = \sum_{t=0}^{|\mathbf{a}_A(\xi)|-1} \sum_{c \in C^{\text{in}}} \mathbf{1}\big[a^A_{t,c}(\xi) \neq \perp\big].$$

For a nonempty finite stimulus set Ξ, let

$$n^A_{\text{txn}}(\Xi) = \sum_{\xi \in \Xi} n^A_{\text{txn}}(\xi), \qquad n^A_{\text{mis}}(\Xi) = \sum_{\xi \in \Xi} n^A_{\text{mis}}(\xi).$$

The accepted-transaction count is activity telemetry rather than a correctness condition.

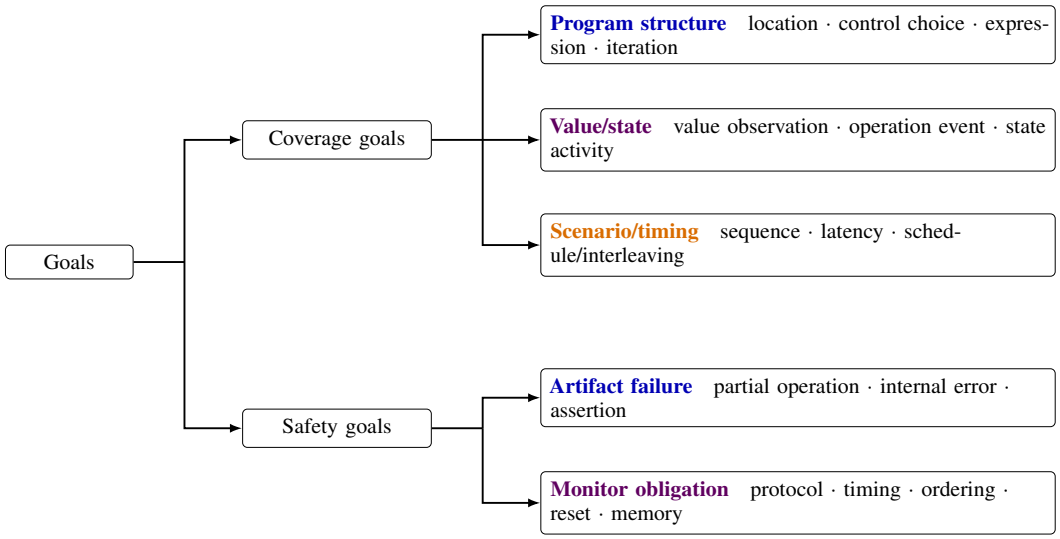


Figure 12: **Goal taxonomy and catalogue sources.** Coverage goals name cases to exercise, while safety goals name bad events to avoid. Colors indicate default schema sources: blue denotes automatic derivation from the artifact or protocol adapter, orange task declarations, and purple either source.

A.6 Goals, Coverage and Scoring

Overview. Output matching checks whether an artifact agrees with $B^\star$ on the generated stimuli. That comparison concerns only the observed outputs and does not characterize what occurs during those executions. We therefore examine which execution cases the stimuli exercise and which bad events they trigger. This subsection defines and classifies those cases and events in a finite catalogue.

Definition 6 (Goals)**.** For each target $T \in \{B^\star, \hat{B}, \hat{R}\}$, let Γ_T be a finite catalogue of named events fixed before execution. A goal is *declared* when it is added to this catalogue, whether it is derived automatically or supplied explicitly by the task. An execution of T *hits* a goal $\gamma \in \Gamma_T$ exactly when the event named by γ occurs during that execution. Every goal belongs to exactly one class:

$$\Gamma_T = \Gamma_T^{\mathrm{cov}} \cup \Gamma_T^{\mathrm{safe}}, \qquad \Gamma_T^{\mathrm{cov}} \cap \Gamma_T^{\mathrm{safe}} = \emptyset.$$

A coverage goal names an execution case that testing should exercise. A safety goal names a bad event that should not occur, either a designated artifact failure or a violation detected by an attached monitor. The catalogue of $B^\star$ is used only to guide stimulus generation. The catalogues of $\hat{B}$ and $\hat{R}$ are evaluated and reported separately.

Goal families. Figure 12 groups goals by class and family, with finite instance rules and hit conditions for each schema given in Table 5. A finite monitor is a finite-state observer that emits goals when a declared pattern completes or an obligation is violated. Tasks and providers may add finite instances or define new schemas by specifying their instance rules and hit conditions.

Catalogue construction. Before testing, BEHAVE-Sim fixes a *goal policy* recording each schema's status, finite identifiers and hit rules. Structural goals are identified at source sites, with scopes, instances and call occurrences resolved by the compiler. Each enabled schema contributes every prescribed instance, regardless of hits. Optimization and loop unrolling do not redefine the catalogue.

Definition 7 (Goal coverage)**.** For a nonempty finite stimulus set Ξ for target T, let $\mathrm{Hit}_T(\Xi) \subseteq \Gamma_T$ be the union of goals observed in the concrete executions of T under all $\xi \in \Xi$. For any nonempty set $G \subseteq \Gamma_T^{\mathrm{cov}}$, define

$$\mathrm{cov}_T(\Xi; G) = \frac{|\mathrm{Hit}_T(\Xi) \cap G|}{|G|}.$$

Choosing G as the goals of one family or schema gives its coverage. The safety goals observed in these executions are $\mathrm{Hit}_T(\Xi) \cap \Gamma_T^{\mathrm{safe}}$.

Table 5: **Standard goal schemas.** Each row defines a finite instance rule and hit condition for Behavior or RTL.

Family	Goal schema	Eligible instances	Goal is hit when
Program structure	location: execution-site entry	each admitted executable source site of the Behavior or RTL artifact	execution enters any occurrence of that site
Program structure	control choice: binary branch or multiway selection outcome	the true and false branches of each admitted binary decision, and each alternative of an admitted multiway selection	execution takes that branch or selects that alternative
Program structure	expression: Boolean-expression evaluation pattern	each policy-generated partial Boolean assignment to the terms of an admitted expression	the expression is evaluated, and every term named by the pattern is evaluated with the specified value
Program structure	iteration: loop zero- or non-zero activation	admitted run-time loops, including procedural RTL loops but excluding generate loops	on a dynamic entry, control exits before body entry for the zero goal, or takes the first body-entry edge for the non-zero goal
Value/state	value observation: state/value bin	a sampling point and a total Boolean predicate over one or more values sampled there	the sampling point occurs and the predicate is true
Value/state	operation event: semantic predicate	an admitted operation site and a total predicate over its operands, result and semantic flags	the operation occurs and the predicate is true
Value/state	state activity: transition or toggle	a sampling point and a Boolean predicate over two consecutive observations, with one bit and one direction fixed for a bit-toggle goal	at the second observation, the predicate holds for the preceding and current values
Scenario/timing	sequence: event or transaction sequence	a task-declared finite monitor over the artifact's transaction or event alphabet	the monitor emits the named goal when an event completes the sequence
Scenario/timing	latency: transaction-latency bin	a task-declared event pair, pairing rule, timebase and finite latency bins	on a matched end event, the count from its paired start in the declared timebase lies in the named bin
Scenario/timing	schedule/interleaving: finite pattern	a finite monitor over schedule-entry event sets and a declared finite order or simultaneity pattern	the monitor emits the named goal when the observed entries complete the pattern
Artifact failure	failure transition	partial operations and internal failures of the admitted Behavior semantics, or embedded errors and assertions designated by the admitted RTL semantics	the designated failure transition executes and emits the named goal
Monitor obligation	obligation violation	an attached finite obligation monitor over observed Behavior or RTL execution, including task-supplied assertions	the monitor emits the named violation goal when the obligation is violated

Definition 8 (Artifact score)**.** For $A \in \{\hat{B}, \hat{R}\}$, define its score on a nonempty finite stimulus set Ξ with completed checks as

$$\mathrm{score}_A(\Xi) = \mathbf{1}\big[n^A_{\mathrm{mis}}(\Xi) = 0 \wedge \mathrm{Hit}_A(\Xi) \cap \Gamma^{\mathrm{safe}}_A = \emptyset\big] .$$

Training reward. The rollout reward is the mean of the two artifact scores,

$$r = \frac{\mathrm{score}_{\hat{B}}(\Xi_{\hat{B}}) + \mathrm{score}_{\hat{R}}(\Xi_{\hat{R}})}{2}.$$

Common coverage metrics. Definition 7 recovers branch, expression and toggle coverage by taking G to be, respectively, the control-choice, Boolean-expression evaluation-pattern and directional bit-toggle goals. FSM state and transition coverage use state-bin and state-transition goals, covergroup coverage uses its declared value-observation goals, and a finite cover property uses a scenario/timing monitor goal. Separately, we report RTL line coverage: the fraction of executable RTL source lines exercised by the stimuli. It is coarser than location-goal coverage because all execution sites on the same source line count as one item.

A.7 Goal-Guided Exploration and Evaluation

Overview. Stimulus generation combines random testing with solver-guided exploration (Godefroid et al., 2005; Sen et al., 2005; Majumdar & Sen, 2007). For each $A \in \{\hat{B}, \hat{R}\}$, evaluation has two phases. First, isolated tests run from reset: the baseline is executed before reference-directed and self-directed searches generate and execute further stimuli. Second, compatible sequences form continuous workloads that preserve state across sequence boundaries. Results from both phases are aggregated using the score and coverage definitions in Subsection A.6.

Search baseline. Let $\mathcal{S}_A$ denote legal isolated tests with bounded input queues and finite execution limits. The nonempty baseline $\Xi^A_{\text{base}} \subseteq \mathcal{S}_A$ contains edge-case patterns and seeded pseudorandom sequences. Search uses baseline and generated stimuli as seeds for further exploration.

Search targets. For $T \in \{B^\star, A\}$, $T \to A$ denotes search on T to generate stimuli for execution on A. Let $\Delta_{T\to A} \subseteq \Gamma_T$ denote goals whose local encodings are enabled and represent the corresponding catalogue hit rules. Excluded goals remain in the catalogue and may still be hit during execution.

Definition 9 (Artifact replay)**.** Let $\mathcal{R}_A$ be the type of raw execution records for A. For a generated artifact A and any legal finite test ξ, artifact replay is the map

$$\textsc{Replay}: \quad (A, \xi) \longmapsto \big(\mathbf{a}_A(\xi), \mathbf{o}_A(\xi), e_A(\xi)\big).$$

For a runtime request, Replay follows its execution policy and retains both the request and resolved stimulus in $e_A(\xi)$. Here, $\xi = (\mathbf{i}, \sigma)$ uses the schedule recorded during execution. $e_A(\xi) \in \mathcal{R}_A$ records the sampled state, inputs, outputs and goal hits together with a record-completeness flag, an artifact outcome and a domain identifier. The flag states whether the full scheduled execution was recorded. The outcome distinguishes normal completion from an artifact failure. $\mathbf{a}_A(\xi)$ and $\mathbf{o}_A(\xi)$ are the accepted input trace and output histories derived from that record as in Subsection A.5. The concrete evaluation stores the collection of execution records

$$\mathcal{E}_A := \{(\xi, e_A(\xi)) \mid \xi \in \Xi_A\}.$$

Execution feedback. Let $\mathcal{M}_A$ store execution traces and goal hits for A and $B^\star$, together with the results of completed output comparisons and monitor checks. Each search obtains goal hits by reusing a suitable execution record or running the stimulus and saving its record. We denote this operation by $\text{Observe}_{T\to A}(\xi; \mathcal{M}_A)$. It returns observed goal hits $J \subseteq \Gamma_T$ from a complete execution of T or a trusted prefix ending in an artifact failure. If neither is available, it returns $\perp$. Each direction maintains its own hit set $H_{T\to A} \subseteq \Gamma_T$, initialized from baseline hits and updated only by execution feedback. The unhit searchable goals are

$$U_{T\to A} = \Delta_{T\to A} \setminus H_{T\to A}.$$

Definition 10 (Solver-guided provider)**.** Fix $A \in \{\hat{B}, \hat{R}\}$, the generated artifact to be evaluated, and $\mathcal{S}_A$ as above. For $T \in \{B^\star, A\}$, a *solver-guided provider* is a triple

$$\mathsf{Prov}_{T\to A} = \big(\Delta_{T\to A}, \text{Select}_{T\to A}, \text{Probe}_{T\to A}\big).$$

Given a seed $\xi = (\mathbf{i}, \sigma) \in \mathcal{S}_A$ and unhit targets $U \subseteq \Delta_{T\to A}$, $\text{Select}_{T\to A}(\xi, U)$ uses the seed's complete execution record of T to construct a finite, deterministically ordered list of solver requests. The list may be empty. Each request q determines a nonempty target set

$$\emptyset \neq G(q) \subseteq U$$

and a nonempty set $F(q)$ of transaction-field positions in $\mathbf{i}$ that the query may change. Before each probe, $\textsc{Restrict}(q, U)$ removes any targets hit after q was selected, without changing its input fields or execution context. It returns $\perp$ if no targets remain. Let $\mathcal{Q}_{T\to A}(\xi)$ denote the requests

that $\mathrm{Select}_{T\to A}$ can produce for seed ξ, including versions with reduced but nonempty target sets. The probe takes a seed $\xi \in \mathcal{S}_A$ and a request $q \in \mathcal{Q}_{T\to A}(\xi)$. Let $u_F = (u_f)_{f\in F(q)}$ be symbolic replacement values for these fields, and let $\mathbf{h}_q = (h_\gamma)_{\gamma\in G(q)}$ contain one Boolean hit bit for each target goal. The provider constructs a finite symbolic-execution relation

$$\mathcal{H}_{T\to A}(\xi, q, u_F, \mathbf{h}_q)$$

in which h_γ is true exactly when the encoded local execution of T hits γ. The probe asks the solver to satisfy

$$\mathsf{Query}_{T\to A}(\xi, q) := \mathcal{H}_{T\to A}(\xi, q, u_F, \mathbf{h}_q) \ \wedge \bigvee_{\gamma\in G(q)} h_\gamma$$

over u_F, $\mathbf{h}_q$ and any provider-specific auxiliary execution variables in $\mathcal{H}_{T\to A}$. Each local query fixes the recorded schedule σ. For a model ν, let $\xi[F(q) \leftarrow \nu(u_F)]$ denote the test obtained by replacing only the fields in $F(q)$. Fixed-schedule tests retain σ. Runtime requests retain their execution policy, so their realized schedules may differ. Then

$$\mathrm{Probe}_{T\to A}(\xi, q) = \begin{cases} \xi[F(q) \leftarrow \nu(u_F)], & \text{if } \nu \models \mathsf{Query}_{T\to A}(\xi, q) \text{ and is decodable,} \\ \bot, & \text{otherwise.} \end{cases}$$

Search budget. The baseline is executed and checked before the search clock starts. Both searches share a wall-clock budget β, including provider preparation, selection, solver calls and candidate execution. Reference-directed search runs up to the budget midpoint, followed by self-directed search using the remaining time. Baseline evaluation, completion of missing isolated checks and continuous workloads use separate finite execution limits outside the search budget.

Isolated exploration. Algorithm 1 initializes each direction's hit set from the baseline records, then alternates solver-guided stimulus generation with execution from reset, with scheduled backpressure where supported. Only seeds with complete execution records are used to select requests. Search stops when its targets or worklist are exhausted, its provider is unavailable, or its deadline is reached, retaining the candidates and records. Write

$$\Xi_{T\to A} = \textsc{Explore}\big(A, T, \Xi^A_{\mathrm{base}}, \mathsf{Prov}_{T\to A}, d; \mathcal{M}_A\big)$$

for the new stimuli retained by that direction.

Continuous workloads. After both searches, the isolated test pool is

$$\Xi^{\mathrm{iso}}_A = \Xi^A_{\mathrm{base}} \cup \Xi_{B^\star\to A} \cup \Xi_{A\to A}.$$

For each $\xi \in \Xi^{\mathrm{iso}}_A$, $\textsc{ExecuteAndCheck}(A, B^\star, \xi; \mathcal{M}_A)$ reuses records and check results for the same artifacts, stimulus and evaluation configuration. It performs only missing execution, output comparisons and monitor checks, and stores the results in $\mathcal{M}_A$. We then construct

$$\Xi^{\mathrm{cont}}_A = \textsc{Continuous}(\Xi^{\mathrm{iso}}_A).$$

Continuous groups compatible sequences in a fixed order, preserving the task's initial-state, interface and event requirements. Sequences that cannot be combined are kept as isolated tests. Each workload is evaluated as a new stimulus using ExecuteAndCheck, with A and $B^\star$ initialized once and state preserved between sequences. Expected outputs come from running $B^\star$ continuously, not concatenating isolated-test results. By default, output receivers remain ready.

Result aggregation. The final evaluation pool is

$$\Xi_A = \Xi^{\mathrm{iso}}_A \cup \Xi^{\mathrm{cont}}_A.$$

Completed records from both phases provide the output mismatches and artifact goal hits used by the score and coverage definitions in Subsection A.6. Algorithm 2 gives the full procedure, returning Ξ_A, $\mathrm{score}_A(\Xi_A)$ and the execution records $\mathcal{E}_A$. The procedure below assumes completed checks. Evaluation-budget interruptions are handled as described in Appendix C.2.

Provider instantiations. Behavior and RTL instantiate $\mathrm{Select}_{T\to A}$ and $\mathcal{H}_{T\to A}$ from Definition 10. They share the solve-and-update step using Z3 (de Moura & Bjørner, 2008).

Behavior provider. The Behavior provider handles $B^\star \to \hat{B}$, $B^\star \to \hat{R}$ and $\hat{B} \to \hat{B}$. For $B^\star \to A$, the provider extracts the accepted input trace $\mathbf{a}_A(\xi)$ from A's execution record and collects goal hits

Algorithm 1: EXPLORE: Search from baseline execution records

Input: artifact A, target $T \in \{B^\star, A\}$, baseline $\Xi^A_{\text{base}} \subseteq \mathcal{S}_A$, provider $\mathsf{Prov}_{T\to A}$, deadline d, shared record store $\mathcal{M}_A$
Output: new candidate set $\Xi_{T\to A} \subseteq \mathcal{S}_A$, with $\mathcal{M}_A$ updated in place

```
// On expiry, retain candidates and records already obtained.
```

Seen $\leftarrow \Xi^A_{\text{base}}$;
Work $\leftarrow$ QUEUE(Ξ^A_{base});
$\Xi_{T\to A} \leftarrow \emptyset$;
$H \leftarrow$ baseline hits of T from complete records or trusted failure prefixes in $\mathcal{M}_A$;
$U \leftarrow \Delta_{T\to A} \setminus H$;
while Work $\neq \emptyset$ *and* $U \neq \emptyset$ *and* NOW() $< d$ **do**
  $\xi \leftarrow$ POP(Work);
  if *no complete execution record of* T *for* ξ *in* $\mathcal{M}_A$ **then**
    continue;
  foreach $q \in \text{Select}_{T\to A}(\xi, U)$ *while* NOW() $< d$ **do**
    $q \leftarrow$ RESTRICT(q, U);
    if $q \neq \perp$ **then**
      $\xi' \leftarrow \text{Probe}_{T\to A}(\xi, q)$;
      if $\xi' \neq \perp$ *and* $\xi' \notin$ Seen **then**
        Seen $\leftarrow$ Seen $\cup \{\xi'\}$;
        PUSH(Work, ξ');
        $\Xi_{T\to A} \leftarrow \Xi_{T\to A} \cup \{\xi'\}$;
        $J \leftarrow \text{Observe}_{T\to A}(\xi'; \mathcal{M}_A)$;
        if $J \neq \perp$ **then**
          $H \leftarrow H \cup J$;
        $U \leftarrow \Delta_{T\to A} \setminus H$;
return $\Xi_{T\to A}$;

and local model states from $B^\star$ running on $\pi(\sigma, \mathbf{a}_A(\xi))$. For $\hat{B} \to \hat{B}$, the same information comes directly from $\hat{B}$. Local queries use these states and retain the transaction fields' source positions in $\mathbf{i}$. In each equation, Φ_q is the symbolic execution constraint over replacement values u_F, and $\psi_{\gamma,q}$ is the Boolean condition that the execution hits $\gamma \in G(q)$. The hit relations are

$$\mathcal{H}_{B^\star\to A}(\xi, q, u_F, \mathbf{h}_q) \equiv \Phi_q^{B^\star\to A}(u_F) \ \wedge \bigwedge_{\gamma\in G(q)} \big(h_\gamma \leftrightarrow \psi_{\gamma,q}^{B^\star\to A}(u_F)\big), \qquad A \in \{\hat{B}, \hat{R}\},$$

and

$$\mathcal{H}_{\hat{B}\to\hat{B}}(\xi, q, u_F, \mathbf{h}_q) \equiv \Phi_q^{\hat{B}\to\hat{B}}(u_F) \ \wedge \bigwedge_{\gamma\in G(q)} \big(h_\gamma \leftrightarrow \psi_{\gamma,q}^{\hat{B}\to\hat{B}}(u_F)\big).$$

RTL provider. An enabled RTL provider handles $\hat{R} \to \hat{R}$. The seed's execution record supplies $(\mathbf{i}, \sigma)$ and, at each schedule boundary k, the state $s_k^{\text{dyn}} = (r_k, \eta_k, z_k)$, comprising adapter bookkeeping, RTL state and monitor state. Each request q selects indices $0 \le k \le \ell < |\sigma|$, the replayed state s_k^{dyn} before $\sigma_{k:\ell}$, target goals $\emptyset \neq G(q) \subseteq U$ and mutable fields $F(q)$ first presented in that segment. The query varies only $F(q)$, fixes the schedule and every other field, and requires the modified execution prefix to reproduce s_k^{dyn} at boundary k. The input queues $\mathbf{i}$ are excluded from this state constraint. Let $\Phi_q^{\hat{R}\to\hat{R}}(u_F)$ be the Yosys-generated RTL segment constraint (Wolf & Glaser, 2013), composed with the protocol adapter and finite monitors, with internal execution variables existentially quantified. Let $\psi_{\gamma,q}^{\hat{R}\to\hat{R}}(u_F)$ encode the catalogue hit rule for $\gamma \in G(q)$ over the segment. For source goals, this requires a source-event encoding in addition to the circuit constraint. The RTL hit relation is

$$\mathcal{H}_{\hat{R}\to\hat{R}}(\xi, q, u_F, \mathbf{h}_q) \equiv \Phi_q^{\hat{R}\to\hat{R}}(u_F) \ \wedge \bigwedge_{\gamma\in G(q)} \big(h_\gamma \leftrightarrow \psi_{\gamma,q}^{\hat{R}\to\hat{R}}(u_F)\big).$$

Algorithm 2: GENERATEANDEVALUATE: Two-sided search and two-phase evaluation

Input: reference $B^\star$, artifact A, baseline Ξ^A_{base}, providers $\mathsf{Prov}_{B^\star \to A}$ and $\mathsf{Prov}_{A \to A}$, total search time $\beta \geq 0$
Output: stimulus set Ξ_A, score $\text{score}_A(\Xi_A)$, execution records $\mathcal{E}_A$

$\mathcal{M}_A \leftarrow \emptyset$;
// Phase 1: evaluate the baseline, then start search timing.
foreach $\xi \in \Xi^A_{\text{base}}$ **do**
  EXECUTEANDCHECK$(A, B^\star, \xi; \mathcal{M}_A)$;
$t_0 \leftarrow$ NOW(); $t_{\text{end}} \leftarrow t_0 + \beta$;
$\Xi_{B^\star \to A} \leftarrow$ EXPLORE$(A, B^\star, \Xi^A_{\text{base}}, \mathsf{Prov}_{B^\star \to A}, t_0 + \beta/2; \mathcal{M}_A)$;
$\Xi_{A \to A} \leftarrow$ EXPLORE$(A, A, \Xi^A_{\text{base}}, \mathsf{Prov}_{A \to A}, t_{\text{end}}; \mathcal{M}_A)$;
$\Xi^{\text{iso}}_A \leftarrow \Xi^A_{\text{base}} \cup \Xi_{B^\star \to A} \cup \Xi_{A \to A}$;
// Complete isolated checks outside the search deadline.
foreach $\xi \in \Xi^{\text{iso}}_A$ **do**
  EXECUTEANDCHECK$(A, B^\star, \xi; \mathcal{M}_A)$;
// Phase 2: construct and evaluate continuous workloads.
$\Xi^{\text{cont}}_A \leftarrow$ CONTINUOUS(Ξ^{iso}_A);
foreach $\xi \in \Xi^{\text{cont}}_A$ **do**
  EXECUTEANDCHECK$(A, B^\star, \xi; \mathcal{M}_A)$;
// Aggregate completed records from both phases.
$\Xi_A \leftarrow \Xi^{\text{iso}}_A \cup \Xi^{\text{cont}}_A$;
$\mathcal{E}_A \leftarrow \{(\xi, e_A(\xi)) \mid \xi \in \Xi_A\}$ from $\mathcal{M}_A$;
return $\big(\Xi_A, \text{score}_A(\Xi_A), \mathcal{E}_A\big)$;

B BEHAVE-SIM: FORMAL SEMANTICS AND BOUNDED GUARANTEES

Purpose. Appendix A evaluates artifacts on a finite stimulus pool. We add optional reachability analysis over a fixed bounded domain. Given an exact Behavior or RTL instance, we use replay-audited traces and solver queries to classify coverage and safety goals as reached, bounded-unreachable or unresolved. These statuses supplement evaluation without changing its stimulus set or score.

Overview. Subsections B.1 through B.3 define the common formalism for bounded trace domains, finite representations, exact encodings, replay audits and classification. Subsections B.4 through B.6 and Subsections B.7 through B.9 construct the RTL and Behavior instances. Subsection B.10 pairs each instance with its evaluated pool and states the reported conclusions. The final subsection states the trusted boundary, claim scope and conclusion. Figure 13 summarizes these dependencies.

B.1 TRACE DOMAINS AND GOAL REACHABILITY

Overview. The analysis considers executions only up to a fixed bound and under fixed task constraints. This subsection gives Behavior and RTL a common semantics for those executions, defines when they reach a declared goal, and embeds declared failures as terminal states.

Definition 11 (Instrumented semantics)**.** An *instrumented semantics* is

$$\mathcal{M} = (X, U, Y, I, \delta, \Gamma),$$

where X is the state space, $\emptyset \neq I \subseteq X$ is the initial-state set, U and Y are the input and output sets, and Γ is the full finite goal catalogue. Its total step function is

$$\delta : X \times U \longrightarrow X \times Y \times 2^\Gamma.$$

Writing $\delta(x, u) = (x', y, H)$, a step from x under input u moves to x', emits y, and hits exactly the goals in H. The Behavior and RTL instances define separately what counts as one step. In both instances, X also contains any finite adapter, harness and monitor state, and H records all goal hits produced during the step. If early termination is represented, the instantiation also designates terminal states $X^{\text{h}} \subseteq X$ and a padding output $\perp_Y \in Y$ such that

$$\delta(x_{\text{h}}, u) = (x_{\text{h}}, \perp_Y, \emptyset) \qquad \text{for every } x_{\text{h}} \in X^{\text{h}},\ u \in U.$$

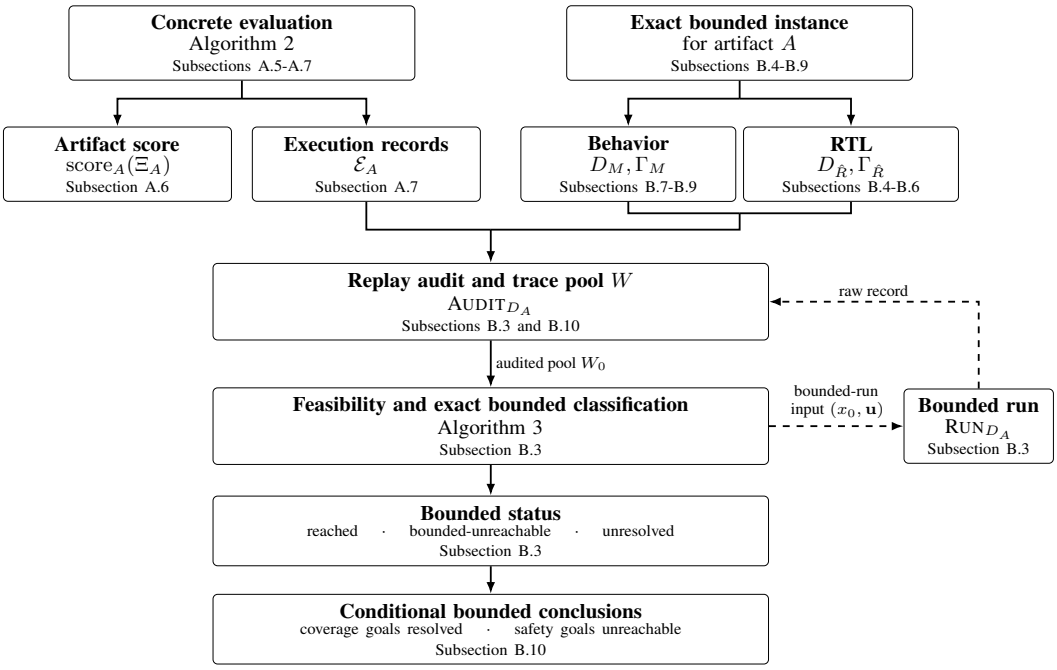


Figure 13: **Concrete evaluation and bounded analysis.** Concrete evaluation returns the artifact score and raw records. With an exact instance, replay-audited traces initialize W_0. The dashed arrows show the validation path for a bounded-run input: artifact replay produces a raw record, which must pass audit before it may update W. This path is used for feasibility initialization and every decoded solver witness. Behavior statuses are mapped back to source goals before reporting. Each bounded conclusion is reported only when its premises hold. Classification does not change the artifact score.

Early termination is therefore padded to the horizon by stuttering steps.

Definition 12 (Fallible step and absorption)**.** Let X, U, Y and Err be sets, let Γ be finite, and fix $\emptyset \neq I^0 \subseteq X$. A *fallible step* is a total function

$$\delta^0 : X \times U \longrightarrow \big(X \times Y \times 2^{\Gamma}\big) \cup \{\mathsf{Err}(\varepsilon, H) \mid \varepsilon \in Err,\ H \subseteq \Gamma\}.$$

Its result is either a completed step (x', y, H) or a failure $\mathsf{Err}(\varepsilon, H)$. Regard the errors as terminal states disjoint from X, renaming them if necessary. For a fresh $\bot_Y \notin Y$, define

$$\bar{X} = X \cup Err,$$
$$\bar{Y} = Y \cup \{\bot_Y\}.$$

The *absorption* of δ^0 at I^0 is

$$\mathrm{Abs}_{I^0}(\delta^0) = (\bar{X}, U, \bar{Y}, I^0, \bar{\delta}, \Gamma),$$

where

$$\bar{\delta}(x, u) = \begin{cases} (x', y, H) & \delta^0(x, u) = (x', y, H), \\ (\varepsilon, \bot_Y, H) & \delta^0(x, u) = \mathsf{Err}(\varepsilon, H), \end{cases}$$

and

$$\bar{\delta}(\varepsilon, u) = (\varepsilon, \bot_Y, \emptyset) \qquad (\varepsilon \in Err).$$

Thus this is an instrumented semantics of Definition 11 with initial set I^0 and terminal set Err.

Definition 13 (Bounded trace domain and reachability)**.** Write $\mathsf{Bool} = \{\mathsf{false}, \mathsf{true}\}$. To define bounded reachability, fix a horizon $N \in \mathbb{N}_{>0}$ and an admissibility predicate

$$K_N : X^{N+1} \times U^N \times Y^N \longrightarrow \mathsf{Bool}$$

that holds exactly for the state, input and output sequences allowed by the fixed task and artifact configuration. The horizon and admissibility predicate are fixed before artifact replay or solver analysis. The *bounded trace domain* is

$$D = (\mathcal{M}, N, K_N).$$

The set of admitted N-step traces is

$$\mathrm{Tr}(D) := \big\{(\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{H}) \;\big|\; \mathbf{x}\in X^{N+1},\ \mathbf{u}\in U^N,\ \mathbf{y}\in Y^N,\ \mathbf{H}\in(2^\Gamma)^N,\\ x_0\in I,\quad K_N(\mathbf{x},\mathbf{u},\mathbf{y})=\mathsf{true},\\ \delta(x_t,u_t)=(x_{t+1},y_t,H_t)\text{ for every } t<N\big\}.$$

Write $w=(\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{H})$ for an admitted trace in $\mathrm{Tr}(D)$. It contains $N+1$ states and one input, output and hit set per step. Define

$$\mathrm{Hit}(w) := \bigcup_{t<N} H_t, \qquad \mathrm{Hit}(W) := \bigcup_{w\in W}\mathrm{Hit}(w) \quad \text{for } W\subseteq \mathrm{Tr}(D).$$

Here Hit takes a trace or a set of traces, whereas $\mathrm{Hit}_T(\Xi)$ in Appendix A.6 takes a stimulus set. The target is fixed by $\mathcal{M}$, so we omit its subscript. The goals reachable within D are

$$\mathrm{Reach}_D := \{\gamma\in\Gamma \mid \exists(\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{H})\in\mathrm{Tr}(D),\ \exists t<N \text{ such that } \gamma\in H_t\}.$$

An admitted trace w is a *witness* for γ when $\gamma\in\mathrm{Hit}(w)$. The domain D is *feasible* when $\mathrm{Tr}(D)\neq\emptyset$.

B.2 Finite Representations and Exact Encodings

Overview. Exact bounded analysis applies only when the fixed trace domain has a finite layered representation. This subsection defines that representation and its exact bounded encoding. Lemma 2 gives a sufficient condition for finite layers to exist. The RTL and Behavior instances below construct their carriers explicitly.

Definition 14 (Finite layered representation)**.** A *finite layered representation* of D consists of finite state sets $X_t^D\subseteq X$ for $t\le N$, finite input sets $U_t^D\subseteq U$ for $t<N$, and finite output sets $Y_t^D\subseteq Y$ for $t<N$. It also supplies a finite initial carrier I_D satisfying

$$\{x_0 \mid (\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{H})\in\mathrm{Tr}(D)\}\subseteq I_D\subseteq I\cap X_0^D.$$

The remaining trace-support conditions are

$$x_t\in X_t^D\quad(t\le N),\qquad u_t\in U_t^D,\quad y_t\in Y_t^D\quad(t<N)$$

for every $(\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{H})\in\mathrm{Tr}(D)$. The sets must also satisfy the layer-closure condition

$$\delta(x,u)\in X_{t+1}^D\times Y_t^D\times 2^\Gamma \quad \text{for every } x\in X_t^D,\ u\in U_t^D,\ t<N.$$

The corresponding finite layered system is

$$\mathcal{A}_D=\big((X_t^D)_{t\le N},(U_t^D)_{t<N},(Y_t^D)_{t<N},I_D,(\delta\restriction(X_t^D\times U_t^D))_{t<N},\Gamma,K_N\big).$$

Here $f\restriction S$ denotes the restriction of a function f to inputs in S. Its trace set is

$$\mathrm{Tr}(\mathcal{A}_D) := \big\{(\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{H}) \;\big|\; \mathbf{x}\in\prod_{t\le N}X_t^D,\quad \mathbf{u}\in\prod_{t<N}U_t^D,\\ \mathbf{y}\in\prod_{t<N}Y_t^D,\quad \mathbf{H}\in(2^\Gamma)^N,\quad x_0\in I_D,\\ K_N(\mathbf{x},\mathbf{u},\mathbf{y})=\mathsf{true},\quad \delta(x_t,u_t)=(x_{t+1},y_t,H_t)\ (t<N)\big\}.$$

If no such representation can be constructed, bounded analysis of D returns Refused.

Lemma 1 (Trace preservation)**.** *Every finite layered representation of D satisfies*

$$\mathrm{Tr}(\mathcal{A}_D)=\mathrm{Tr}(D).$$

Consequently, its reachable-goal set is Reach_D.

Proof. Fix a finite layered representation of D, and let $w=(\mathbf{x},\mathbf{u},\mathbf{y},\mathbf{H})$. If $w\in\mathrm{Tr}(\mathcal{A}_D)$, then the layer-membership conditions, $x_0\in I_D\subseteq I$, and the shared admissibility and transition conditions give

$$w\in\mathrm{Tr}(\mathcal{A}_D)\Longrightarrow w\in\mathrm{Tr}(D),\qquad \mathrm{Tr}(\mathcal{A}_D)\subseteq\mathrm{Tr}(D).$$

Conversely, let $w \in \mathrm{Tr}(D)$. The trace-support conditions of the fixed finite layered representation give

$$x_t \in X_t^D \ (t \le N), \qquad u_t \in U_t^D, \quad y_t \in Y_t^D \ (t < N),$$

while the initial-carrier condition gives $x_0 \in I_D$. The admissibility and transition conditions already hold, so

$$w \in \mathrm{Tr}(D) \Longrightarrow w \in \mathrm{Tr}(\mathcal{A}_D), \qquad \mathrm{Tr}(D) \subseteq \mathrm{Tr}(\mathcal{A}_D).$$

The two inclusions prove equality. By Definition 13, the reachable-goal set is therefore

$$\big\{\gamma \in \Gamma \;\big|\; \exists(\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{H}) \in \mathrm{Tr}(\mathcal{A}_D),\ \exists t < N : \gamma \in H_t\big\} = \mathrm{Reach}_D. \qquad \square$$

Lemma 2 (Finite-layer construction)**.** *Suppose the set of initial states occurring in* $\mathrm{Tr}(D)$ *is finite and, at every* $t < N$*, only finitely many values occur as* u_t *among traces in* $\mathrm{Tr}(D)$*. Then* D *admits a finite layered representation.*

Proof. Let $I_D = X_0^D$ be the set of initial states occurring in $\mathrm{Tr}(D)$, and set

$$U_t^D = \{u_t \mid (\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{H}) \in \mathrm{Tr}(D)\} \qquad (t < N).$$

Definition 13 gives $I_D \subseteq I$, so the initial-carrier condition holds. Each U_t^D is finite by the hypothesis of the lemma. For $t = 0, \dots, N-1$ in order, apply δ to every pair in $X_t^D \times U_t^D$. Let X_{t+1}^D collect the resulting next states and Y_t^D the resulting outputs. We show by induction that X_t^D is finite and contains every x_t occurring in an admitted trace. The base case is $X_0^D = I_D$. For the induction step, $X_t^D \times U_t^D$ is finite, so its image under δ, and hence X_{t+1}^D and Y_t^D, is finite. For any admitted trace, $u_t \in U_t^D$. Its transition equation and the induction hypothesis then give $x_{t+1} \in X_{t+1}^D$ and $y_t \in Y_t^D$. This establishes trace support. Finally, because the construction includes every $x \in X_t^D$ and $u \in U_t^D$, their result satisfies $\delta(x, u) \in X_{t+1}^D \times Y_t^D \times 2^\Gamma$ for every $t < N$. This is the layer-closure condition. $\square$

Definition 15 (Exact bounded encoding)**.** An *exact bounded encoding* translates the finite layered system $\mathcal{A}_D$ into solver formulas without adding or removing an initial state, admitted sequence, transition or goal hit. Let $\mathbb{B} = \{0, 1\}$. The encoding first supplies injective maps

$$\big(\mathrm{enc}_t^X : X_t^D \to \mathbb{B}^{d_t^X}\big)_{t \le N}, \qquad \big(\mathrm{enc}_t^U : U_t^D \to \mathbb{B}^{d_t^U}\big)_{t < N}, \qquad \big(\mathrm{enc}_t^Y : Y_t^D \to \mathbb{B}^{d_t^Y}\big)_{t < N},$$

where the exponents are positive bit widths. Each enc map converts a semantic value into a bit vector. Write dec_t^X, dec_t^U and dec_t^Y for the respective inverse maps on bit vectors produced by these encodings. Since not every bit vector need encode a value, the following Boolean predicates recognize exactly the valid encodings at each trace position:

$$\begin{aligned}
\mathrm{Valid}_t^X(\widehat{x}) &\iff \widehat{x} \in \{\mathrm{enc}_t^X(x) \mid x \in X_t^D\},\\
\mathrm{Valid}_t^U(\widehat{u}) &\iff \widehat{u} \in \{\mathrm{enc}_t^U(u) \mid u \in U_t^D\},\\
\mathrm{Valid}_t^Y(\widehat{y}) &\iff \widehat{y} \in \{\mathrm{enc}_t^Y(y) \mid y \in Y_t^D\}.
\end{aligned}$$

The encoding also supplies Boolean predicates for the semantic conditions: $\widehat{I}_D$ tests initial-state membership, $\widehat{K}_N$ tests sequence admissibility, and $\widehat{\delta}_t^D$ tests whether an encoded tuple represents a transition at position $t < N$. On every $\widehat{x}$ satisfying Valid_0^X,

$$\widehat{I}_D(\widehat{x}) \iff \mathrm{dec}_0^X(\widehat{x}) \in I_D.$$

Hence Valid_0^X recognizes every encoded state in X_0^D, whereas $\widehat{I}_D$ selects those whose decoded states belong to I_D. For encoded sequences

$$\widehat{\mathbf{x}} = (\widehat{x}_t)_{t \le N}, \qquad \widehat{\mathbf{u}} = (\widehat{u}_t)_{t < N}, \qquad \widehat{\mathbf{y}} = (\widehat{y}_t)_{t < N}$$

whose components satisfy the corresponding position-indexed validity predicates, namely $\mathrm{Valid}_t^X(\widehat{x}_t)$ for $t \le N$ and $\mathrm{Valid}_t^U(\widehat{u}_t)$ and $\mathrm{Valid}_t^Y(\widehat{y}_t)$ for $t < N$,

$$\widehat{K}_N(\widehat{\mathbf{x}}, \widehat{\mathbf{u}}, \widehat{\mathbf{y}}) \iff K_N\Big(\big(\mathrm{dec}_t^X(\widehat{x}_t)\big)_{t \le N}, \big(\mathrm{dec}_t^U(\widehat{u}_t)\big)_{t < N}, \big(\mathrm{dec}_t^Y(\widehat{y}_t)\big)_{t < N}\Big).$$

For every $t < N$, every $\mathbf{h} = (h_\gamma)_{\gamma\in\Gamma} \in \{\mathsf{false}, \mathsf{true}\}^\Gamma$, and every $(\widehat{x}, \widehat{u}, \widehat{x}', \widehat{y})$ satisfying Valid_t^X, Valid_t^U, Valid_{t+1}^X and Valid_t^Y, respectively,

$$\begin{aligned}\widehat{\delta}_t^D(\widehat{x}, \widehat{u}, \widehat{x}', \widehat{y}, \mathbf{h}) \iff \delta\big(\mathrm{dec}_t^X(\widehat{x}), \mathrm{dec}_t^U(\widehat{u})\big) \\ = \big(\mathrm{dec}_{t+1}^X(\widehat{x}'), \mathrm{dec}_t^Y(\widehat{y}), \{\gamma \in \Gamma \mid h_\gamma = \mathsf{true}\}\big).\end{aligned}$$

B.3 Bounded Reachability Analysis

Overview. Subsection A.7 evaluates artifacts on a finite stimulus pool. Here, exact queries analyze goal reachability over the full bounded domain $\mathrm{Tr}(D)$. The goal set selected for classification is $G \subseteq \Gamma$, and the full-catalogue analysis takes $G = \Gamma$. Solver answers and replay audits classify each goal as reached, bounded-unreachable or unresolved.

Definition 16 (Bounded queries)**.** Fix a finite layered representation of D and an exact bounded encoding of $\mathcal{A}_D$. For a solver query, let $\widehat{\mathbf{x}}, \widehat{\mathbf{u}}, \widehat{\mathbf{y}}$ denote its encoded state, input and output variables. For $t < N$, let $\mathbf{h}_t = (h_{t,\gamma})_{\gamma\in\Gamma}$ be fresh Boolean variables encoding the hit set at step t. The base formula encoding an admitted trace in D is

$$\begin{aligned}F_D = \widehat{I}_D(\widehat{x}_0) \wedge \bigwedge_{t\le N} \mathrm{Valid}_t^X(\widehat{x}_t) \\ \wedge \bigwedge_{t<N} \big(\mathrm{Valid}_t^U(\widehat{u}_t) \wedge \mathrm{Valid}_t^Y(\widehat{y}_t)\big) \\ \wedge \bigwedge_{t<N} \widehat{\delta}_t^D(\widehat{x}_t, \widehat{u}_t, \widehat{x}_{t+1}, \widehat{y}_t, \mathbf{h}_t) \wedge \widehat{K}_N(\widehat{\mathbf{x}}, \widehat{\mathbf{u}}, \widehat{\mathbf{y}}).\end{aligned}$$

For a target goal set $S \subseteq \Gamma$, define

$$\Phi_{D,S} = F_D \wedge \bigvee_{\gamma\in S} \bigvee_{t<N} h_{t,\gamma}.$$

The single-goal query is $\Phi_{D,\{\gamma\}}$.

Theorem 2 (Bounded-query correctness)**.** *Given a finite layered representation of D and an exact bounded encoding of $\mathcal{A}_D$,*

$$F_D \text{ is satisfiable} \iff \mathrm{Tr}(D) \neq \emptyset.$$

Thus F_D is satisfiable exactly when D is feasible. For every $S \subseteq \Gamma$,

$$\Phi_{D,S} \text{ is satisfiable} \iff S \cap \mathrm{Reach}_D \neq \emptyset.$$

Every satisfying assignment to F_D decodes to a trace in $\mathrm{Tr}(D)$. Every satisfying assignment to $\Phi_{D,S}$ decodes to one that hits a goal in S.

Proof. For $w = (\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{H}) \in \mathrm{Tr}(D)$, Lemma 1 gives $w \in \mathrm{Tr}(\mathcal{A}_D)$. Define an assignment ν_w by

$$\begin{aligned}\nu_w(\widehat{x}_t) &= \mathrm{enc}_t^X(x_t) && (t \le N),\\ \nu_w(\widehat{u}_t) &= \mathrm{enc}_t^U(u_t), \qquad \nu_w(\widehat{y}_t) = \mathrm{enc}_t^Y(y_t) && (t < N),\\ \nu_w(h_{t,\gamma}) &= \mathsf{true} \iff \gamma \in H_t && (t < N,\ \gamma \in \Gamma).\end{aligned}$$

Exactness of the encoding gives $\nu_w \models F_D$. Conversely, let $\nu \models F_D$. The validity predicates permit every encoded component to be decoded. Set

$$H_t^\nu = \{\gamma \in \Gamma \mid \nu(h_{t,\gamma}) = \mathsf{true}\}.$$

The initial-state, transition and admissibility conjuncts of F_D give a decoded trace $w_\nu \in \mathrm{Tr}(\mathcal{A}_D)$. By Lemma 1, $w_\nu \in \mathrm{Tr}(D)$. Therefore

$$F_D \text{ is satisfiable} \iff \mathrm{Tr}(D) \neq \emptyset.$$

Finally, expanding the hit disjunction and then decoding gives

$$\begin{aligned}\Phi_{D,S} \text{ is satisfiable} &\iff \exists \nu,\ \exists t < N,\ \exists \gamma \in S : \nu \models F_D \ \wedge\ \nu(h_{t,\gamma}) = \mathsf{true}\\ &\iff \exists (\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{H}) \in \mathrm{Tr}(D),\ \ \exists t < N,\ \exists \gamma \in S : \gamma \in H_t\\ &\iff S \cap \mathrm{Reach}_D \neq \emptyset.\end{aligned}$$

The constructions of ν_w and w_ν also prove the decoding claims. □

Definition 17 (Replay audit and evaluated pool)**.** Fix a generated artifact A together with its analysis domain D, a finite layered representation and an exact bounded encoding. The replay audit is

$$\textsc{Audit}_D : \mathcal{R}_A \longrightarrow \mathrm{Tr}(D) \cup \{\mathsf{Outside}, \mathsf{Inconsistent}, \mathsf{Refused}\}.$$

For a raw record $e \in \mathcal{R}_A$, return Refused if it is incomplete or lacks a valid domain identifier. For a record identifying D, reconstruct $w = (\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{H})$ and require exactly $N+1$ states and N inputs, outputs and hit sets, or return Refused. Records are not truncated or padded. Complete records of artifact failures are audited normally. Return Outside if the record belongs to another domain or, for a record identifying D, a state, input or output lies outside its position-specific carrier, $x_0 \notin I_D$, or $K_N(\mathbf{x}, \mathbf{u}, \mathbf{y}) = \mathsf{false}$. If the record is not outside D, require $H_t \subseteq \Gamma$ for every $t < N$ and define

$$\widehat{x}_t^w := \mathrm{enc}_t^X(x_t) \qquad (t \le N),$$
$$\widehat{u}_t^w := \mathrm{enc}_t^U(u_t), \qquad \widehat{y}_t^w := \mathrm{enc}_t^Y(y_t) \quad (t < N).$$

Let $\mathbf{h}_t^w = (h_{t,\gamma}^w)_{\gamma \in \Gamma}$, where $h_{t,\gamma}^w = \mathsf{true}$ iff $\gamma \in H_t$. Each step must satisfy

$$\widehat{\delta}_t^D(\widehat{x}_t^w, \widehat{u}_t^w, \widehat{x}_{t+1}^w, \widehat{y}_t^w, \mathbf{h}_t^w) \qquad (t < N).$$

A failed requirement returns Inconsistent. Otherwise exactness gives $w \in \mathrm{Tr}(D)$, and the audit returns w. Audit every record in $\mathcal{E}_A$. Any Inconsistent or Refused result stops bounded classification, while Outside records are discarded. If no audit stops classification, define

$$W_{\mathrm{eval}}^A := \left\{ w \in \mathrm{Tr}(D) \;\middle|\; \textsc{Audit}_D(e) = w \text{ for some } (\xi, e) \in \mathcal{E}_A \right\}.$$

Bounded-run interface. Each artifact supplies

$$\textsc{Run}_D : I_D \times \prod_{t<N} U_t^D \longrightarrow \mathcal{R}_A \cup \{\mathsf{Refused}\},$$

where $\textsc{Run}_D(x_0, \mathbf{u})$ realizes x_0 as an artifact-specific concrete state, runs A for N steps under $\mathbf{u}$, and returns the resulting raw record. It returns Refused if no complete raw record can be produced.

Initial audited pool. After the preceding audit processes every evaluation record without returning Refused or Inconsistent, if $W_{\mathrm{eval}}^A \neq \emptyset$, set $W_0 = W_{\mathrm{eval}}^A$. Otherwise, solve F_D. By Theorem 2, an unsatisfiable result establishes $\mathrm{Tr}(D) = \emptyset$, while an unresolved result leaves feasibility unresolved. From a satisfying assignment, decode $x_0 \in I_D$ and $\mathbf{u} \in \prod_{t<N} U_t^D$, then invoke $\textsc{Run}_D(x_0, \mathbf{u})$. A refused run is reported as Refused. Any returned raw record is passed to $\textsc{Audit}_D$. If the audit returns $w_{\mathrm{f}} \in \mathrm{Tr}(D)$, set $W_0 = \{w_{\mathrm{f}}\}$. A refused audit is reported as Refused. An Outside or Inconsistent result is reported as Inconsistent.

Goal classification. Given a nonempty W_0 and the selected set G, Algorithm 3 marks the goals hit by W_0 as reached and queries subsets of the remaining goals. A satisfiable query marks goals reached only after the artifact is run on the decoded values and the resulting record passes the audit. An unsatisfiable query marks its target set as bounded-unreachable. If a query over several goals is unresolved, the algorithm retries smaller subsets. Only an unresolved singleton is marked unresolved. The result is (Rch, Unr, Unk, W), where the first three sets contain the reached, bounded-unreachable and unresolved goals, and W is the audited pool witnessing Rch.

Theorem 3 (Bounded-status soundness)**.** *Fix a bounded trace domain D, a finite layered representation of D with an exact bounded encoding, a goal set $G \subseteq \Gamma$, and a nonempty audited pool $W_0 \subseteq \mathrm{Tr}(D)$. Assume that every solver call returns a satisfying model, a correct unsatisfiable result or an unresolved result, and that every decoding, $\textsc{Run}_D$ invocation and audit terminates. Then $\textsc{ClassifyGoals}(G, W_0)$ in Algorithm 3 terminates. If it returns (Rch, Unr, Unk, W), its three status sets partition G and*

$$Rch \subseteq \mathrm{Reach}_D, \qquad Unr \cap \mathrm{Reach}_D = \emptyset,$$

and every goal in Rch has a witness in W.

Proof. At the head of each outer-loop iteration, Rch, Unr and Unk contain the goals currently marked reached, bounded-unreachable and unresolved. The set W is the accumulated pool of audited traces, and $Pend \subseteq G$ contains the goals not yet assigned a status. Maintain

$$W \subseteq \mathrm{Tr}(D), \qquad Rch = \mathrm{Hit}(W) \cap G,$$

Algorithm 3: CLASSIFYGOALS: Replay-audited bounded reachability

Input: goal set $G \subseteq \Gamma$, nonempty audited pool $W_0 \subseteq \mathrm{Tr}(D)$
Output: Inconsistent, Refused, or (Rch, Unr, Unk, W)

$W \leftarrow W_0$; $Rch \leftarrow \mathrm{Hit}(W_0) \cap G$
$Unr \leftarrow \emptyset$; $Unk \leftarrow \emptyset$; $Pend \leftarrow G \setminus Rch$
while $Pend \neq \emptyset$ **do**
  choose nonempty $S \subseteq Pend$
  solve $\Phi_{D,S}$
  while *the result is unresolved and* $|S| > 1$ **do**
    choose a nonempty proper subset $S' \subsetneq S$
    $S \leftarrow S'$
    solve $\Phi_{D,S}$
  if *satisfiable* **then**
    decode the initial state and input sequence from the satisfying model ν
    invoke RUN_D on the decoded values
    if *it returns* Refused **then**
      return Refused
    let e be the returned raw record
    $r_{\mathrm{aud}} \leftarrow \mathrm{AUDIT}_D(e)$
    if $r_{\mathrm{aud}} =$ Refused **then**
      return Refused
    if $r_{\mathrm{aud}} \in \{$Outside, Inconsistent$\}$ **then**
      return Inconsistent
    $w \leftarrow r_{\mathrm{aud}}$
    if $\mathrm{Hit}(w) \cap S = \emptyset$ *or* $\mathrm{Hit}(w) \cap Unr \neq \emptyset$ **then**
      return Inconsistent
    $W \leftarrow W \cup \{w\}$
    $Rch \leftarrow Rch \cup \big(\mathrm{Hit}(w) \cap G\big)$
    $Unk \leftarrow Unk \setminus \mathrm{Hit}(w)$
    $Pend \leftarrow Pend \setminus \mathrm{Hit}(w)$
  else if *unsatisfiable* **then**
    $Unr \leftarrow Unr \cup S$
    $Pend \leftarrow Pend \setminus S$
  else // unresolved singleton
    $Unk \leftarrow Unk \cup S$
    $Pend \leftarrow Pend \setminus S$
return (Rch, Unr, Unk, W)

and require Rch, Unr, Unk and $Pend$ to be pairwise disjoint with union G. These properties hold initially. Within one outer-loop iteration, $S \subseteq Pend$ is the nonempty goal set targeted by the current query. Each unresolved inner-loop iteration replaces S with a nonempty proper subset, so the inner loop terminates. For a satisfiable query, a successful audit gives $w \in \mathrm{Tr}(D)$. Let $J = \mathrm{Hit}(w) \cap G$. The checks give $J \cap S \neq \emptyset$ and $J \cap Unr = \emptyset$. The updates add w to W, move the goals in $J \cap (Pend \cup Unk)$ to Rch, and preserve the invariant. They also strictly decrease $|Pend|$ because $J \cap S \neq \emptyset$ and $S \subseteq Pend$. For an unsatisfiable query, Theorem 2 gives $S \cap \mathrm{Reach}_D = \emptyset$. Moving the nonempty set S from $Pend$ to Unr therefore preserves the partition, adds only bounded-unreachable goals and strictly decreases $|Pend|$. If the result remains unresolved after the inner loop, then $|S| = 1$. Moving this singleton from $Pend$ to Unk again preserves the invariant and decreases $|Pend|$. Any failed check returns immediately. Since G is finite, the outer loop terminates. At a normal return, $Pend = \emptyset$, so the three status sets partition G. For any γ, the invariant gives

$$\gamma \in Rch \quad \Longrightarrow \quad \gamma \in \mathrm{Hit}(W) \quad \Longrightarrow \quad \exists w \in W \text{ such that } \gamma \in \mathrm{Hit}(w).$$

Since $W \subseteq \mathrm{Tr}(D)$, this w is a witness within D, and Definition 13 gives $\gamma \in \mathrm{Reach}_D$. Finally, every set added to Unr was proved disjoint from Reach_D by Theorem 2. Hence $Unr \cap \mathrm{Reach}_D = \emptyset$. □

Corollary 1 (Complete bounded classification)**.** *Under the hypotheses of Theorem 3, suppose Algorithm 3 returns normally with $Unk = \emptyset$. Then*

$$Rch = \mathrm{Reach}_D \cap G, \qquad Unr = G \setminus \mathrm{Reach}_D.$$

Proof. Since $Unk = \emptyset$, Theorem 3 gives $G = Rch \cup Unr$, with the two sets disjoint. Consider any $\gamma \in \mathrm{Reach}_D \cap G$. The partition places γ in Rch or Unr, and $Unr \cap \mathrm{Reach}_D = \emptyset$ excludes the latter. Hence $\mathrm{Reach}_D \cap G \subseteq Rch$. Conversely, if $\gamma \in Rch$, then Theorem 3 gives $\gamma \in \mathrm{Reach}_D$, while the partition gives $\gamma \in G$. Thus $Rch \subseteq \mathrm{Reach}_D \cap G$, proving the first equality. The partition then gives

$$\begin{aligned} Unr &= G \setminus Rch \\ &= G \setminus (\mathrm{Reach}_D \cap G) \\ &= G \setminus \mathrm{Reach}_D. \end{aligned}$$

□

B.4 RTL Event-Slot Semantics and Protocol Adapters

Overview. This subsection instantiates the instrumented semantics of Definition 11 for pin-level RTL. It defines the RTL transition for one event slot. Behavior consumes and emits logical transactions directly. RTL instead uses a task-declared protocol adapter to drive interface pins and identify the transactions accepted and emitted in each event slot.

Goal producers. The coverage/safety partition of Definition 6 classifies goals by purpose. Independently, the RTL catalogue has the producer partition

$$\Gamma_{\hat{R}} = \Gamma_{\hat{R}}^{\mathrm{slot}} \cup \Gamma_{\hat{R}}^{\mathrm{mon}}, \qquad \Gamma_{\hat{R}}^{\mathrm{slot}} \cap \Gamma_{\hat{R}}^{\mathrm{mon}} = \emptyset.$$

The event-slot semantics emits only slot goals, and the finite monitors emit only monitor goals. Either producer class may contain coverage and safety goals.

Definition 18 (RTL event-slot semantics)**.** Let $\mathcal{E}_R$ be the finite set of legal clock and reset event sets. Each event is tagged by its signal, domain and polarity. Let Q be the finite RTL-state set, let U_R and Y_R be the finite pin-input and pin-observation sets, and let $\emptyset \neq I_R \subseteq Q$ be the initial-state set. The slot semantics is the total function

$$\delta_{\hat{R}}^{\mathrm{slot}} : Q \times U_R \times \mathcal{E}_R \longrightarrow Q \times Y_R \times 2^{\Gamma_{\hat{R}}^{\mathrm{slot}}}.$$

One application of $\delta_{\hat{R}}^{\mathrm{slot}}$ is an *RTL event slot*. Each slot is one scheduled step: inputs and resets are driven and settled before pin sampling and active clock events. Writing $\delta_{\hat{R}}^{\mathrm{slot}}(\eta_t, u_t^{\mathrm{pin}}, e_t) = (\eta_{t+1}, o_t, H_t^{\mathrm{slot}})$, o_t is sampled before active clock events, η_{t+1} is the settled state after clocks return to inactive levels, and H_t^{slot} contains exactly the raw RTL goals hit at declared sampling phases. N counts slots, not internal phases. In the single-clock measurement profile, each slot spans one cycle.

Definition 19 (Finite protocol adapter)**.** Fix the transaction interface of Definition 1 and the RTL event-slot semantics of Definition 18. A finite protocol adapter is a tuple

$$\Pi = (X_\Pi, I_\Pi, \mathcal{A}_\Pi, \lambda_\Pi, \tau_\Pi)$$

selected by the task's endpoint bindings. These bindings declare each endpoint's pins, their ownership, and its clock and reset domain. An adapter state has the form $d = (\mathbf{i}, r) \in X_\Pi$, where $\mathbf{i}$ is an input-queue family permitted by the fixed bounds of $\mathcal{S}_{\hat{R}}$ and r is finite environment and protocol bookkeeping. When the task declares a finite external-memory model, its state is a component of r, as shown in Figure 11. The queues remain fixed during a run, while r may change. Let $I_\Pi \subseteq X_\Pi$ be the nonempty set of states with initial bookkeeping. For a stimulus $\xi = (\mathbf{i}, \sigma)$, initialization selects $d_0 = (\mathbf{i}, r_0) \in I_\Pi$. For RTL, each schedule entry introduced in Definition 2 has the form

$$\sigma_t = (e_t, \alpha_t) \in \Sigma_\Pi := \mathcal{E}_R \times \mathcal{A}_\Pi.$$

Here $\mathcal{A}_\Pi$ is the finite environment-action set. The component e_t contains the clock and reset events, while α_t specifies which queued inputs become available for presentation, which outputs to accept, any external-memory response, and other environment choices. The total drive function

$$\lambda_\Pi : X_\Pi \times \Sigma_\Pi \longrightarrow U_R, \qquad u_t^{\mathrm{pin}} = \lambda_\Pi(d_t, \sigma_t)$$

constructs the RTL pin inputs from the queued transactions, current bookkeeping, slot events and environment choices. After the RTL produces the pin observation $o_t \in Y_R$, the adapter updates its state and identifies the transfers in that slot. Let

$$\mathcal{B}^{\mathrm{in}} = \prod_{c \in C^{\mathrm{in}}} (T_c \cup \{\bot\}), \qquad \mathcal{B}^{\mathrm{out}} = \prod_{c \in C^{\mathrm{out}}} (T_c \cup \{\bot\}),$$

where each component is the transaction accepted or emitted on that endpoint, or $\bot$ if there is none. The total update function is

$$\tau_\Pi : X_\Pi \times \Sigma_\Pi \times U_R \times Y_R \longrightarrow X_\Pi \times \mathcal{B}^{\mathrm{in}} \times \mathcal{B}^{\mathrm{out}},$$

with

$$(d_{t+1}, a_t, b_t) = \tau_\Pi(d_t, \sigma_t, u_t^{\mathrm{pin}}, o_t).$$

Writing $a_{t,c}$ for the c-component of a_t, the non-$\bot$ entries of $(a_{t,c})_t$ form a prefix of $\mathbf{i}_c$. Thus the adapter preserves queue order and reports no input transaction more than once.

Ready/valid instance. The endpoint binding losslessly packs each transaction $v \in T_c$ onto its payload pins and unpacks each observed payload, with $unpack_c(pack_c(v)) = v$. Figure 11 shows the signal ownership. For an input endpoint c, λ_Π drives in_valid and in_data, while τ_Π observes the RTL-driven in_ready. The bookkeeping contains a pointer $p_{c,t} \in \{0, \ldots, k_c\}$ to the first unaccepted input, initially zero, and tracks the inputs made available by α_t that remain pending. For each input endpoint, the adapter presents the oldest pending transaction and holds it stable until acceptance. Reset updates pending work according to the task's declared reset policy. Let $present_{c,t}$ indicate that a pending transaction is being presented. The driven signals are

$$in_valid_{c,t} = present_{c,t} \wedge [p_{c,t} < k_c], \qquad in_data_{c,t} = \begin{cases} pack_c(x_{c,p_{c,t}}) & in_valid_{c,t}, \\ 0 & \text{otherwise}, \end{cases}.$$

For an output endpoint, λ_Π drives out_ready from α_t, while τ_Π observes the RTL-driven out_valid and out_data. At an active, non-reset event for the corresponding endpoint, the input and output transfer conditions are

$$\begin{aligned} fire_{c,t}^{\mathrm{in}} &\iff in_valid_{c,t} \wedge in_ready_{c,t} \qquad (c \in C^{\mathrm{in}}), \\ fire_{c,t}^{\mathrm{out}} &\iff out_valid_{c,t} \wedge out_ready_{c,t} \qquad (c \in C^{\mathrm{out}}). \end{aligned}$$

No transfer occurs for that endpoint in any other slot. An input transfer puts $x_{c,p_{c,t}}$ in a_t and advances the pointer. An output transfer puts $unpack_c(out_data_{c,t})$ in b_t. A non-transferring endpoint contributes $\bot$, and a reset applies the adapter's declared reset update. If reset occurs with transactions in flight, the task declares whether they are cancelled or remain outstanding. Without this declaration, checks that depend on them remain unresolved. The adapter retains a presented input transaction until transfer unless a reset intervenes. Thus, for every input endpoint c and $t < N_{\hat{R}} - 1$ such that neither e_t nor e_{t+1} resets c,

$$present_{c,t} \wedge [p_{c,t} < k_c] \wedge \neg fire_{c,t}^{\mathrm{in}} \implies present_{c,t+1}.$$

The dual design obligation retains an output while it is stalled. For every output endpoint c and $t < N_{\hat{R}} - 1$ such that neither e_t nor e_{t+1} resets c,

$$out_valid_{c,t} \wedge \neg out_ready_{c,t} \implies out_valid_{c,t+1} \wedge out_data_{c,t+1} = out_data_{c,t}.$$

The corresponding monitor emits a safety goal if this implication fails.

Transaction projection. During an RTL replay of ξ, write the per-slot batches returned by τ_Π as

$$a_t^{\hat{R}}(\xi) = \big(a_{t,c}^{\hat{R}}(\xi)\big)_{c \in C^{\mathrm{in}}}, \qquad b_t^{\hat{R}}(\xi) = \big(b_{t,c}^{\hat{R}}(\xi)\big)_{c \in C^{\mathrm{out}}}.$$

The accepted input trace is $\mathbf{a}_{\hat{R}}(\xi) = (a_t^{\hat{R}}(\xi))_{t < N_{\hat{R}}}$. For each output endpoint c, $o_{\hat{R},c}(\xi)$ is obtained from $(b_{t,c}^{\hat{R}}(\xi))_{t < N_{\hat{R}}}$ by removing the $\bot$ entries. The output-history family is $\mathbf{o}_{\hat{R}}(\xi) = (o_{\hat{R},c}(\xi))_{c \in C^{\mathrm{out}}}$.

B.5 RTL Closed-Loop Semantics and Trace Domain

Overview. Using the event-slot semantics and protocol adapter of Subsection B.4, this subsection instantiates the instrumented semantics and bounded trace domain of Subsection B.1 for RTL. It composes the adapter, RTL design and enabled finite monitors, then defines the allowed clock, reset and environment-action sequences.

Definition 20 (Finite monitor product)**.** Let Z be the finite state set of the product of all monitors enabled for $\hat{R}$, let $z_0 \in Z$ be its initial state, and let its total update function be

$$\mu : Z \times \Sigma_\Pi \times Q \times U_R \times Y_R \times Q \times \mathcal{B}^{\text{in}} \times \mathcal{B}^{\text{out}} \longrightarrow Z \times 2^{\Gamma^{\text{mon}}_{\hat{R}}} .$$

It observes the schedule entry, the RTL step and the adapter's transaction events, then returns the next monitor state and the monitor goals hit in that slot. Its components emit the enabled coverage and safety goals assigned to monitors, and their reset behavior is part of μ. An optional task-declared response deadline pairs an accepted input with its first response offer and emits a safety goal if the declared bound expires first. If the task declares reset restoration, another component tracks the declared reset sequence. It emits a safety goal when the declared post-reset condition fails at completion, and a scenario goal when the sequence completes after at least one non-reset slot. If no monitor is enabled, then $Z = \{z_0\}$ and μ emits no goals.

Definition 21 (Closed-loop RTL semantics)**.** Define

$$\begin{aligned} X^{\text{cl}}_{\hat{R}} &= X_\Pi \times Q \times Z, \\ Y^{\text{cl}}_{\hat{R}} &= Y_R \times \mathcal{B}^{\text{in}} \times \mathcal{B}^{\text{out}}, \\ I^{\text{cl}}_{\hat{R}} &= I_\Pi \times I_R \times \{z_0\}. \end{aligned}$$

The instrumented semantics used to analyze $\hat{R}$ is

$$\mathcal{M}_{\hat{R}} = \big(X^{\text{cl}}_{\hat{R}}, \Sigma_\Pi, Y^{\text{cl}}_{\hat{R}}, I^{\text{cl}}_{\hat{R}}, \delta^{\text{cl}}_{\hat{R}}, \Gamma_{\hat{R}}\big),$$

where, for one slot, $d_t \in X_\Pi$, $\eta_t \in Q$, $z_t \in Z$, $\sigma_t = (e_t, \alpha_t)$, and $u^{\text{pin}}_t = \lambda_\Pi(d_t, \sigma_t)$,

$$\begin{aligned} (\eta_{t+1}, o_t, H^{\text{slot}}_t) &= \delta^{\text{slot}}_{\hat{R}}(\eta_t, u^{\text{pin}}_t, e_t), \\ (d_{t+1}, a_t, b_t) &= \tau_\Pi(d_t, \sigma_t, u^{\text{pin}}_t, o_t), \\ (z_{t+1}, H^{\text{mon}}_t) &= \mu(z_t, \sigma_t, \eta_t, u^{\text{pin}}_t, o_t, \eta_{t+1}, a_t, b_t), \\ H_t &= H^{\text{slot}}_t \cup H^{\text{mon}}_t, \\ \delta^{\text{cl}}_{\hat{R}}((d_t, \eta_t, z_t), \sigma_t) &= ((d_{t+1}, \eta_{t+1}, z_{t+1}), (o_t, a_t, b_t), H_t). \end{aligned}$$

Definition 22 (RTL trace domain)**.** Set $N = N_{\hat{R}}$. Let K^{env}_N be the admissibility predicate of Definition 13 for $\mathcal{M}_{\hat{R}}$. It holds exactly for the N-slot closed-loop traces whose clock and reset events satisfy the fixed harness configuration and whose environment actions satisfy the obligations of the selected protocol adapter. Define

$$D_{\hat{R}} = (\mathcal{M}_{\hat{R}}, N, K^{\text{env}}_N).$$

An environment-side violation therefore excludes the trace from $\text{Tr}(D_{\hat{R}})$, whereas a design-side protocol violation is recorded as a safety hit.

B.6 RTL Finite Representation, Exact Encoding and Audits

Overview. This subsection constructs the finite RTL representation and exact encoding, states the required semantic, catalogue and replay fidelity assumptions, and defines the catalogue and replay audits.

Admitted RTL. The RTL analysis admits elaborated designs whose fixed-width signals and fixed-size memories range over a finite logic domain. It requires an explicit initial-state predicate and a total deterministic event-slot transition. The selected protocol adapter and monitors must likewise be finite and total.

Component encoding. Assign fixed injective encodings and exact validity predicates to the components of $X^{\mathrm{cl}}_{\hat{R}}$, Σ_Π and $Y^{\mathrm{cl}}_{\hat{R}}$, and to the internal pin-input set U_R. At slot t, write the product encodings as

$$\widehat{x}_t = (\widehat{d}_t, \widehat{\eta}_t, \widehat{z}_t), \qquad \widehat{\sigma}_t = (\widehat{e}_t, \widehat{\alpha}_t), \qquad \widehat{y}_t = (\widehat{o}_t, \widehat{a}_t, \widehat{b}_t).$$

A product encoding is valid exactly when each component encoding is valid. Exact analysis requires the RTL frontend to supply the encoded initial-state predicate $\widehat{I}_R$ and the one-slot relation

$$\widehat{\delta}^{\mathrm{slot}}_{\hat{R}}(\widehat{\eta}, \widehat{u}^{\mathrm{pin}}, \widehat{e}, \widehat{\eta}', \widehat{o}, \mathbf{h}^{\mathrm{slot}}), \qquad \mathbf{h}^{\mathrm{slot}} = (h^{\mathrm{slot}}_\gamma)_{\gamma \in \Gamma^{\mathrm{slot}}_{\hat{R}}} \in \mathsf{Bool}^{\Gamma^{\mathrm{slot}}_{\hat{R}}}.$$

The hit bits must encode the catalogue hit rules from Subsection A.6. For source goals, this includes preserving their event semantics through compilation. Here $\widehat{u}^{\mathrm{pin}}$ encodes a value in U_R. The adapter and monitor functions in Definitions 19 and 20, and K^{env}_N, have finite domains and therefore admit exact encodings. Write these as $\widehat{\lambda}_\Pi$, $\widehat{\tau}_\Pi$, $\widehat{\mu}$ and $\widehat{K}^{\mathrm{env}}_N$. On valid encodings, the first three return the encoded results of λ_Π, τ_Π and μ, while $\widehat{K}^{\mathrm{env}}_N$ holds exactly for traces satisfying K^{env}_N. For $\widehat{x} = (\widehat{d}, \widehat{\eta}, \widehat{z})$, define the product initial-state predicate by

$$\widehat{I}_{D_{\hat{R}}}(\widehat{x}) \quad \Longleftrightarrow \quad d \in I_\Pi \ \wedge\ \widehat{I}_R(\widehat{\eta}) \ \wedge\ z = z_0,$$

where d and z are the values decoded from $\widehat{d}$ and $\widehat{z}$. The monitor hit set is represented by $\mathbf{h}^{\mathrm{mon}} = (h^{\mathrm{mon}}_\gamma)_{\gamma \in \Gamma^{\mathrm{mon}}_{\hat{R}}} \in \mathsf{Bool}^{\Gamma^{\mathrm{mon}}_{\hat{R}}}$.

Assumption 2 (RTL semantic fidelity)**.** Under the event-slot semantics of Definition 18, every valid encoding $\widehat{\eta}$ of η satisfies

$$\widehat{I}_R(\widehat{\eta}) \quad \Longleftrightarrow \quad \eta \in I_R.$$

For every valid encoding of $\eta, u^{\mathrm{pin}}, e, \eta', o$ and every $\mathbf{h}^{\mathrm{slot}} \in \mathsf{Bool}^{\Gamma^{\mathrm{slot}}_{\hat{R}}}$,

$$\begin{aligned}&\widehat{\delta}^{\mathrm{slot}}_{\hat{R}}(\widehat{\eta}, \widehat{u}^{\mathrm{pin}}, \widehat{e}, \widehat{\eta}', \widehat{o}, \mathbf{h}^{\mathrm{slot}})\\ &\quad\Longleftrightarrow \quad \delta^{\mathrm{slot}}_{\hat{R}}(\eta, u^{\mathrm{pin}}, e) = \left(\eta', o, \{\gamma \in \Gamma^{\mathrm{slot}}_{\hat{R}} \mid h^{\mathrm{slot}}_\gamma = \mathsf{true}\}\right).\end{aligned}$$

Theorem 4 (Exact RTL encoding)**.** *Under Assumption 2 and the exact component encodings fixed above, set*

$$X^{D_{\hat{R}}}_t = X^{\mathrm{cl}}_{\hat{R}} \quad (t \le N), \qquad U^{D_{\hat{R}}}_t = \Sigma_\Pi, \quad Y^{D_{\hat{R}}}_t = Y^{\mathrm{cl}}_{\hat{R}} \quad (t < N), \qquad I_{D_{\hat{R}}} = I^{\mathrm{cl}}_{\hat{R}}.$$

For $t < N$, let $\mathbf{h}_t = (h_{t,\gamma})_{\gamma \in \Gamma_{\hat{R}}}$, set $\widehat{u}^{\mathrm{pin}}_t = \widehat{\lambda}_\Pi(\widehat{d}_t, \widehat{\sigma}_t)$, and define

$$\begin{aligned}&\widehat{\delta}^{D_{\hat{R}}}_t(\widehat{x}_t, \widehat{\sigma}_t, \widehat{x}_{t+1}, \widehat{y}_t, \mathbf{h}_t)\\ &\Longleftrightarrow \exists (\mathbf{h}^{\mathrm{slot}}_t, \mathbf{h}^{\mathrm{mon}}_t) \in \mathsf{Bool}^{\Gamma^{\mathrm{slot}}_{\hat{R}}} \times \mathsf{Bool}^{\Gamma^{\mathrm{mon}}_{\hat{R}}} :\\ &\qquad \widehat{\delta}^{\mathrm{slot}}_{\hat{R}}(\widehat{\eta}_t, \widehat{u}^{\mathrm{pin}}_t, \widehat{e}_t, \widehat{\eta}_{t+1}, \widehat{o}_t, \mathbf{h}^{\mathrm{slot}}_t)\\ &\qquad \wedge\ (\widehat{d}_{t+1}, \widehat{a}_t, \widehat{b}_t) = \widehat{\tau}_\Pi(\widehat{d}_t, \widehat{\sigma}_t, \widehat{u}^{\mathrm{pin}}_t, \widehat{o}_t)\\ &\qquad \wedge\ (\widehat{z}_{t+1}, \mathbf{h}^{\mathrm{mon}}_t) = \widehat{\mu}(\widehat{z}_t, \widehat{\sigma}_t, \widehat{\eta}_t, \widehat{u}^{\mathrm{pin}}_t, \widehat{o}_t, \widehat{\eta}_{t+1}, \widehat{a}_t, \widehat{b}_t)\\ &\qquad \wedge \bigwedge_{\gamma \in \Gamma^{\mathrm{slot}}_{\hat{R}}} \left(h_{t,\gamma} \leftrightarrow h^{\mathrm{slot}}_{t,\gamma}\right)\\ &\qquad \wedge \bigwedge_{\gamma \in \Gamma^{\mathrm{mon}}_{\hat{R}}} \left(h_{t,\gamma} \leftrightarrow h^{\mathrm{mon}}_{t,\gamma}\right).\end{aligned}$$

These carriers form a finite layered representation of $D_{\hat{R}}$, and the component encodings, validity predicates, $\widehat{I}_{D_{\hat{R}}}$, $\widehat{K}^{\mathrm{env}}_N$ and $(\widehat{\delta}^{D_{\hat{R}}}_t)_{t<N}$ form an exact bounded encoding of $\mathcal{A}_{D_{\hat{R}}}$.

Proof. The displayed carriers are finite by Definitions 1, 18, 19 and 20. Because they are the full carriers of $\mathcal{M}_{\hat{R}}$, every trace in $\mathrm{Tr}(D_{\hat{R}})$ satisfies trace support, and $I_{D_{\hat{R}}} = I^{\mathrm{cl}}_{\hat{R}}$ satisfies the initial-carrier condition. Totality of $\delta^{\mathrm{cl}}_{\hat{R}}$ gives layer closure. Hence the displayed carriers form a finite layered representation. Fix $t < N$ and valid encodings of $x_t, \sigma_t, x_{t+1}, y_t$. Exactness of $\widehat{\lambda}_\Pi$, Assumption 2, and exactness of $\widehat{\tau}_\Pi$ and $\widehat{\mu}$ give the three component updates in Definition 21. Assumption 2 identifies

the slot hit vector with H_t^{slot}, while exactness of $\widehat{\mu}$ identifies the monitor hit vector with H_t^{mon}. The producer partition gives their union H_t. Therefore

$$\widehat{\delta}_t^{D_{\hat{R}}}(\widehat{x}_t, \widehat{\sigma}_t, \widehat{x}_{t+1}, \widehat{y}_t, \mathbf{h}_t) \iff \delta_{\hat{R}}^{\mathrm{cl}}(x_t, \sigma_t) = \left(x_{t+1}, y_t, \{\gamma \in \Gamma_{\hat{R}} \mid h_{t,\gamma} = \mathsf{true}\}\right).$$

This is the transition equivalence required by Definition 15. For valid encoded sequences, write $\widehat{\mathbf{x}}, \widehat{\sigma}, \widehat{\mathbf{y}}$ for the encoded values and $\mathbf{x}, \sigma, \mathbf{y}$ for their decodings. The definition of $\widehat{I}_{D_{\hat{R}}}$ and Assumption 2 give the first equivalence below. Exactness of $\widehat{K}_N^{\mathrm{env}}$ gives the second:

$$\widehat{I}_{D_{\hat{R}}}(\widehat{x}_0) \iff x_0 \in I_{D_{\hat{R}}}, \qquad \widehat{K}_N^{\mathrm{env}}(\widehat{\mathbf{x}}, \widehat{\sigma}, \widehat{\mathbf{y}}) \iff K_N^{\mathrm{env}}(\mathbf{x}, \sigma, \mathbf{y}).$$

Together with the component validity predicates, these are all the conditions of Definition 15. □

Assumption 3 (RTL catalogue fidelity)**.** The RTL instrumentation and monitors implement exactly the identifiers and hit rules fixed by the goal policy. For $\gamma \in \Gamma_{\hat{R}}^{\mathrm{slot}}$, h_γ^{slot} is true exactly when an RTL occurrence assigned to γ activates. For $\gamma \in \Gamma_{\hat{R}}^{\mathrm{mon}}$, h_γ^{mon} is true exactly when the corresponding monitor emits γ.

Assumption 4 (RTL replay fidelity)**.** For every $x_0 \in I_{\hat{R}}^{\mathrm{cl}}$ and $\sigma \in \Sigma_\Pi^N$, if $\textsc{Run}_{D_{\hat{R}}}(x_0, \sigma)$ returns a raw record, that record is a complete and faithful account of the actual N-slot closed-loop replay from x_0 under σ.

Catalogue and replay audits. The RTL harness instantiates $\textsc{Run}_{D_{\hat{R}}}$ by restoring the selected initial product state and executing all N schedule entries. Its record includes the adapter, RTL and monitor states, pin observations, transaction events and both component hit sets, including reset and no-transfer slots. Before bounded analysis, the catalogue audit checks that the complete encoded goal identifiers and hit rules match $\Gamma_{\hat{R}}$. Each raw record is then passed to $\textsc{Audit}_{D_{\hat{R}}}$ of Definition 17 using $\widehat{\delta}_t^{D_{\hat{R}}}$. Only records returned as traces are retained.

B.7 Behavior IR and Procedure Semantics

Overview. RTL advances through event slots and uses a protocol adapter to recover logical transactions from pins. A Behavior model already operates on transactions, so one step is one complete model invocation. The RTL event-slot transition is a semantic primitive, whereas the Behavior transition is derived from the small IR below. This subsection defines the IR's state updates, outputs, goal hits and declared failures.

Definition 23 (Behavior IR)**.** The semantic types are

$$\tau ::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{array}[n, \tau], \qquad n \in \mathbb{N}_{>0},$$

with value sets

$$[\![\mathsf{bool}]\!] = \mathsf{Bool}, \qquad [\![\mathsf{int}]\!] = \mathbb{Z}, \qquad [\![\mathsf{array}[n, \tau]]\!] = [\![\tau]\!]^n.$$

Thus arrays have fixed length and value semantics. Let $\mathcal{T}$ be the set of semantic types and Var the set of variable names. A typed context is a finite partial map $\Theta : \mathsf{Var} \rightharpoonup \mathcal{T}$, and $\mathrm{dom}(\Theta)$ is the finite set of variables it declares. Its total typed valuations form the set

$$\mathrm{Val}(\Theta) = \left\{ \rho : \mathrm{dom}(\Theta) \to \bigcup_{\tau \in \mathcal{T}} [\![\tau]\!] \;\middle|\; \rho(z) \in [\![\Theta(z)]\!] \text{ for every } z \in \mathrm{dom}(\Theta) \right\}.$$

Thus each $\rho \in \mathrm{Val}(\Theta)$ maps every declared variable to a value of its declared type. A Behavior procedure M represents one complete model invocation and consists of

$$M = (\Theta_S, \Theta_I, \Theta_L, \Theta_O, c, U_M, Y_M, Err_M, \Gamma_M^{\mathrm{cov}}, \Gamma_M^{\mathrm{safe}}).$$

The error names form a finite set Err_M. The finite goal catalogue is

$$\Gamma_M = \Gamma_M^{\mathrm{cov}} \cup \Gamma_M^{\mathrm{safe}}, \qquad \Gamma_M^{\mathrm{cov}} \cap \Gamma_M^{\mathrm{safe}} = \emptyset.$$

Its expressions and commands are

$$\begin{aligned} e &::= z \mid \mathit{lit} \mid e_1[e_2] \mid p(e_1, \ldots, e_m), \\ c &::= \mathsf{skip} \mid z := e \mid z[e_1] := e_2 \mid c_1; c_2 \\ &\quad \mid \mathsf{if}\ e\ \mathsf{then}\ c_T\ \mathsf{else}\ c_F \mid \mathsf{for}\ j < e\ \mathsf{do}\ c \\ &\quad \mid \mathsf{mark}(\gamma) \mid \mathsf{fail}(\varepsilon, \gamma_s). \end{aligned}$$

Here z ranges over variables, lit over typed literals, p over registered primitives, and j over the read-only indices introduced by for. Moreover, $\gamma \in \Gamma_M$, $\gamma_s \in \Gamma_M^{\mathrm{safe}}$, and $\varepsilon \in \mathit{Err}_M$. The command $\mathsf{mark}(\gamma)$ records γ and continues, while $\mathsf{fail}(\varepsilon, \gamma_s)$ records γ_s and produces the failure $\mathsf{Err}(\varepsilon, \{\gamma_s\})$ of Definition 12. Each primitive has a fixed, typed, total denotation. Bit vectors, enumerations, and fixed- and floating-point values are represented by integers and typed primitives. Records are flattened into typed variables, and finite memories use arrays. Partial source operations, including array reads and indexed updates, are guarded by explicit validity checks whose invalid branches execute fail. The pairwise-disjoint contexts $\Theta_S, \Theta_I, \Theta_L$ and Θ_O declare its persistent state, input, temporary and output variables, respectively. Write $\Theta_M = \Theta_S \cup \Theta_I \cup \Theta_L \cup \Theta_O$ for their union. The command is c, and the finite input and output domains are

$$U_M \subseteq \mathrm{Val}(\Theta_I), \qquad Y_M \subseteq \mathrm{Val}(\Theta_O)\,.$$

We regard each valuation of a context as one structured value. Thus U_M is the finite set of admitted input records, and Y_M is the finite set of normal outputs. Each $y \in Y_M$ represents the complete ordered output sequence of one invocation, including the empty and multi-transaction cases.

Definition 24 (Big-step command and procedure semantics)**.** A partial valuation ρ over Θ_M satisfies $\mathrm{dom}(\rho) \subseteq \mathrm{dom}(\Theta_M)$ and $\rho(z) \in [\![\Theta_M(z)]\!]$ for every $z \in \mathrm{dom}(\rho)$. Write $\rho|_\Theta$ for the restriction of ρ to the variables in a context Θ. For an expression e whose variables belong to $\mathrm{dom}(\rho)$ and whose array accesses are in bounds, let $[\![e]\!]_\rho$ denote its value in ρ:

$$\begin{aligned} [\![z]\!]_\rho &= \rho(z), & [\![\mathit{lit}]\!]_\rho &= \mathsf{val}(\mathit{lit}), \\ [\![e_1[e_2]]\!]_\rho &= \mathsf{select}([\![e_1]\!]_\rho, [\![e_2]\!]_\rho), & [\![p(e_1, \ldots, e_m)]\!]_\rho &= [\![p]\!]([\![e_1]\!]_\rho, \ldots, [\![e_m]\!]_\rho). \end{aligned}$$

Here $\mathsf{val}(\mathit{lit})$ denotes the value of a typed literal, $\mathsf{select}(a, i)$ returns the i-th element of a, and $[\![p]\!]$ is the fixed interpretation of the registered primitive p. The judgment

$$\langle c, \rho \rangle \Downarrow R$$

relates an initial valuation to the result of executing the complete command c. A result is either a normal result $\langle \rho', H \rangle$ or a failure $\mathsf{Err}(\varepsilon, H)$, where $\varepsilon \in \mathit{Err}_M$ and $H \subseteq \Gamma_M$. The coverage and safety hits are $H \cap \Gamma_M^{\mathrm{cov}}$ and $H \cap \Gamma_M^{\mathrm{safe}}$, respectively. For an array a, a valid index k, and a value v, let $\mathsf{update}(a, k, v)$ be a with its k-th element replaced by v. The judgment is the least relation satisfying the following rules. The primitive-command rules are

$$\frac{}{\langle \mathsf{skip}, \rho \rangle \Downarrow \langle \rho, \emptyset \rangle} \qquad \frac{[\![e]\!]_\rho = v}{\langle z := e, \rho \rangle \Downarrow \langle \rho[z \mapsto v], \emptyset \rangle},$$

$$\frac{[\![e_1]\!]_\rho = k \qquad [\![e_2]\!]_\rho = v}{\langle z[e_1] := e_2, \rho \rangle \Downarrow \langle \rho[z \mapsto \mathsf{update}(\rho(z), k, v)], \emptyset \rangle}, \qquad \frac{}{\langle \mathsf{mark}(\gamma), \rho \rangle \Downarrow \langle \rho, \{\gamma\} \rangle},$$

$$\frac{}{\langle \mathsf{fail}(\varepsilon, \gamma_s), \rho \rangle \Downarrow \mathsf{Err}(\varepsilon, \{\gamma_s\})}.$$

Sequential composition has three rules:

$$\frac{\langle c_1, \rho \rangle \Downarrow \langle \rho_1, H_1 \rangle \qquad \langle c_2, \rho_1 \rangle \Downarrow \langle \rho_2, H_2 \rangle}{\langle c_1; c_2, \rho \rangle \Downarrow \langle \rho_2, H_1 \cup H_2 \rangle},$$

$$\frac{\langle c_1, \rho \rangle \Downarrow \mathsf{Err}(\varepsilon, H_1)}{\langle c_1; c_2, \rho \rangle \Downarrow \mathsf{Err}(\varepsilon, H_1)}, \qquad \frac{\langle c_1, \rho \rangle \Downarrow \langle \rho_1, H_1 \rangle \qquad \langle c_2, \rho_1 \rangle \Downarrow \mathsf{Err}(\varepsilon, H_2)}{\langle c_1; c_2, \rho \rangle \Downarrow \mathsf{Err}(\varepsilon, H_1 \cup H_2)}.$$

The conditional rules are

$$\frac{[\![e]\!]_\rho = \mathsf{true} \qquad \langle c_T, \rho \rangle \Downarrow R}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_T\ \mathsf{else}\ c_F, \rho \rangle \Downarrow R}, \qquad \frac{[\![e]\!]_\rho = \mathsf{false} \qquad \langle c_F, \rho \rangle \Downarrow R}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_T\ \mathsf{else}\ c_F, \rho \rangle \Downarrow R}.$$

Thus the guard is evaluated once. For $m = \max(0, [\![e]\!]_\rho)$, let $c^{[m]} = c[j := 0]; \cdots ; c[j := m-1]$, using capture-avoiding substitution and taking $c^{[0]} = \mathsf{skip}$. The loop rule is

$$\frac{m = \max(0, [\![e]\!]_\rho) \qquad \langle c^{[m]}, \rho\rangle \Downarrow R}{\langle \mathsf{for}\ j < e\ \mathsf{do}\ c, \rho\rangle \Downarrow R}.$$

Let $X_M = \mathrm{Val}(\Theta_S)$. For $s \in X_M$ and $i \in U_M$, let $\rho_0 = s \cup i$ be the partial valuation on $\Theta_S \cup \Theta_I$. Locals and outputs are initially undefined. With this execution relation, call M *well formed* when c is well typed under Θ_M, guards are Boolean, loop bounds are integers, and only state, local and output variables may be assigned. For every $s \in X_M$ and $i \in U_M$, every variable read during execution from ρ_0 must be defined and every array access must be in bounds. Each normal result must also satisfy

$$\langle c, \rho_0\rangle \Downarrow \langle \rho_f, H\rangle \quad \Longrightarrow \quad \rho_f|_{\Theta_O} \in Y_M.$$

Define the fallible step induced by M as

$$\delta^0_M(s,i) = \begin{cases} \big(\rho_f|_{\Theta_S}, \rho_f|_{\Theta_O}, H\big) & \langle c, \rho_0\rangle \Downarrow \langle \rho_f, H\rangle, \\ \mathsf{Err}(\varepsilon, H) & \langle c, \rho_0\rangle \Downarrow \mathsf{Err}(\varepsilon, H). \end{cases}$$

Lemma 3 (Total instrumented Behavior semantics)**.** *For every well-formed M,*

$$\forall (s,i) \in X_M \times U_M, \qquad \exists! R :\ \langle c, s \cup i\rangle \Downarrow R.$$

Hence δ^0_M is total. Consequently, for every $\emptyset \neq I^0_M \subseteq X_M$,

$$\mathcal{M}_M(I^0_M) = \mathrm{Abs}_{I^0_M}(\delta^0_M)$$

is an instrumented semantics in the sense of Definition 11. Write $\bar{\delta}_M$ for its step function. Its transitions from a normal state into an error state satisfy

$$\forall s \in X_M,\ \forall i \in U_M,\ \forall \varepsilon \in \mathit{Err}_M,\ \forall H \subseteq \Gamma_M,$$
$$\bar{\delta}_M(s,i) = (\varepsilon, \bot_Y, H) \quad \Longrightarrow \quad H \cap \Gamma^{\mathrm{safe}}_M \neq \emptyset.$$

Proof. Fix $(s,i) \in X_M \times U_M$. For every command d obtained from c by taking subcommands and substituting loop indices, and every valuation ρ reached at d, prove simultaneously by structural induction that

$$\exists! R :\ \langle d, \rho\rangle \Downarrow R,$$
$$\langle d, \rho\rangle \Downarrow \mathsf{Err}(\varepsilon, H) \quad \Longrightarrow \quad H \cap \Gamma^{\mathrm{safe}}_M \neq \emptyset.$$

Well-formedness makes every evaluated expression defined, and its value is unique. The primitive cases are immediate, and fail is the only one producing a failure, with a hit in Γ^{safe}_M. For $d = d_1; d_2$, the unique result of d_1 selects the failure rule or uniquely determines the input to d_2. Union with earlier hits preserves a safety hit. A Boolean guard selects exactly one branch. For a loop, the bound fixes a unique finite m. An inner induction on m, using the structural induction hypothesis for each $c[j := k]$, proves the claim for $c^{[m]}$. Taking $d = c$ and $\rho = s \cup i$ gives a unique result. On normal completion, typing places the state projection in X_M, and well-formedness places the output projection in Y_M. Thus δ^0_M is total, and absorption gives the stated instrumented semantics. Absorption preserves the hit set on the transition from X_M into the corresponding error state, proving the final implication. □

B.8 Behavior Finite Representation and Exact Encoding

Overview. Definition 15 states what an exact bounded encoding must provide. For a well-formed procedure M and a nonempty initial set $I^0_M \subseteq X_M$, let

$$D = (\mathcal{M}_M(I^0_M), N, K_N).$$

This subsection constructs an exact bounded encoding of D using finite sets that contain every reachable state and every intermediate value.

Definition 25 (Behavior representation plan)**.** A *Behavior representation plan* P for (M, D) assigns finite state layers, finite sets of internal values, one encoding for each semantic type, and an exact encoded term for each primitive occurrence, subject to the conditions below.

State layers. For each $t \le N$, the plan contains a finite set $S_t \subseteq X_M$. These sets satisfy

$$\{x_0 \mid (\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{H}) \in \mathrm{Tr}(D)\} \subseteq S_0,$$
$$\{s' \mid \exists s \in S_t,\ u \in U_M,\ y,\ H : \delta_M^0(s, u) = (s', y, H)\} \subseteq S_{t+1} \qquad (t < N).$$

The first condition covers every admitted initial state. The second closes the normal states under one step. The sets S_t contain only normal states. If $\delta_M^0(s, u) = \mathsf{Err}(\varepsilon, H)$, absorption enters the terminal state $\varepsilon \in Err_M$. Program execution then stops. The state ε is repeated only to pad the trace to horizon N. The full state layers are therefore

$$X_t^D := S_t \cup Err_M \qquad (t \le N).$$

Finite internal value sets. For every $t < N$, command or expression occurrence ℓ, and quantity q evaluated at ℓ, let τ_q be the type of q. The plan contains a finite set

$$\mathcal{R}_{t,\ell}(q) \subseteq [\![\tau_q]\!].$$

It contains every value taken by q when a step starts in S_t with an input in U_M, including values computed before a failure. The quantities comprise variables, expression results, array indices and elements, outputs and loop trip counts. A set may be empty when its occurrence is unreachable.

Encodings. For each semantic type τ used by M, collect its finite internal value sets, the component values occurring in S_t, U_M, and Y_M, every typed literal in c, and the finitely many index constants introduced by the planned loop expansions. Their union is finite. The plan gives it one injective fixed-width bit-vector encoding enc_τ, so a value has the same encoding at every program point.

Primitive exactness. For an occurrence of the registered primitive $p(q_1, \dots, q_m)$ at ℓ, where q_j is its j-th operand, write $R_j = \mathcal{R}_{t,\ell}(q_j)$ and $R_{\mathrm{out}} = \mathcal{R}_{t,\ell}(p(q_1, \dots, q_m))$, and let τ_j and τ_{out} be their types. Its semantic result must remain in the planned output set:

$$\{[\![p]\!](v_1, \dots, v_m) \mid (v_1, \dots, v_m) \in R_1 \times \cdots \times R_m\} \subseteq R_{\mathrm{out}}.$$

The plan also provides a bit-vector term $\widehat{p}_{t,\ell}$ satisfying

$$\widehat{p}_{t,\ell}\big(\mathrm{enc}_{\tau_1}(v_1), \dots, \mathrm{enc}_{\tau_m}(v_m)\big) = \mathrm{enc}_{\tau_{\mathrm{out}}}\big([\![p]\!](v_1, \dots, v_m)\big)$$

for every $(v_1, \dots, v_m) \in R_1 \times \cdots \times R_m$. Boolean connectives, array packing and array update use their exact solver operations. The solver term $\mathrm{ite}(b, v_T, v_F)$ returns v_T when b is true and v_F otherwise. A plan is *valid* if all the above conditions hold.

Lemma 4 (Plan-induced finite representation)**.** *Given a valid plan P, put*

$$I_D = I_M^0 \cap S_0,$$
$$U_t^D = U_M, \qquad Y_t^D = Y_M \cup \{\bot_Y\} \quad (t < N).$$

Together with the full state layers $(X_t^D)_{t \le N}$ defined above, these sets and initial carrier form a finite layered representation of D, with layered system $\mathcal{A}_D$. Moreover,

$$\mathrm{Tr}(\mathcal{A}_D) = \mathrm{Tr}(D),$$

so $\mathcal{A}_D$ and D have the same reachable goals.

Proof. Each S_t is finite by validity of P, and Err_M, U_M, and Y_M are finite by Definition 23. Hence every X_t^D, U_t^D, and Y_t^D is finite, and I_D is finite because $I_D \subseteq S_0$. Every trace of D starts in I_M^0, and the initial-layer condition of P places its initial state in S_0. Therefore

$$\{x_0 \mid (\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{H}) \in \mathrm{Tr}(D)\} \subseteq I_D \subseteq I_M^0 \cap X_0^D.$$

Fix $t < N$, $x \in X_t^D$, and $u \in U_t^D = U_M$. If $x \in S_t$, a normally completing step reaches S_{t+1} by the state-layer closure of P and produces an output in Y_M. A failing step reaches some $\varepsilon \in Err_M$ and produces $\bot_Y$. If $x \in Err_M$, absorption keeps the same error state and again produces $\bot_Y$. In every case,

$$\bar{\delta}_M(x, u) \in X_{t+1}^D \times Y_t^D \times 2^{\Gamma_M}.$$

This proves layer closure. Starting from the initial inclusion above, induction on t gives the remaining trace-support conditions. The displayed carriers therefore form a finite layered representation of D. Finally, Lemma 1 gives the trace equality and equality of the reachable-goal sets. □

State, input and output encoding. A normal state, input or output is a typed valuation. Encode it by concatenating the type encodings of its components in their fixed context order. Disjoint tags distinguish normal states from terminal errors and normal outputs from $\perp_Y$. A terminal error also carries an injective encoding of its name. Any otherwise empty boundary encoding receives one fixed padding bit. This gives the injective maps required by Definition 15, with positive bit widths d_t^X, d_t^U, and d_t^Y:

$$\big(\mathrm{enc}_t^X : X_t^D \to \mathbb{B}^{d_t^X}\big)_{t\le N}, \qquad \big(\mathrm{enc}_t^U : U_M \to \mathbb{B}^{d_t^U}\big)_{t<N},$$
$$\big(\mathrm{enc}_t^Y : (Y_M \cup \{\perp_Y\}) \to \mathbb{B}^{d_t^Y}\big)_{t<N}.$$

Let Pack_r^X and Pack_r^Y denote the fixed wiring terms that concatenate the selected component encodings and add the normal-state or normal-output tag. Their defining identities are

$$\mathrm{Pack}_r^X\big((\mathrm{enc}_{\Theta_S(z)}(s(z)))_{z\in\mathrm{dom}(\Theta_S)}\big) = \mathrm{enc}_r^X(s) \quad (r \le N,\ s \in S_r),$$
$$\mathrm{Pack}_r^Y\big((\mathrm{enc}_{\Theta_O(z)}(y(z)))_{z\in\mathrm{dom}(\Theta_O)}\big) = \mathrm{enc}_r^Y(y) \quad (r < N,\ y \in Y_M).$$

Their inverses on their images and their validity predicates are defined as in Definition 15. The semantic initial-state condition is $x_0 \in I_D$. Its encoded form is

$$\widehat{I}_D(\widehat{x}_0) \equiv \bigvee_{s\in I_D} \widehat{x}_0 = \mathrm{enc}_0^X(s).$$

Thus $\widehat{I}_D(\widehat{x}_0)$ holds exactly when the solver variable $\widehat{x}_0$ equals the encoding of some initial state in I_D. Define $\widehat{K}_N$ by enumerating exactly the valid encoded sequences whose decoded sequences satisfy K_N. This enumeration is finite because all carriers are finite.

Definition 26 (Plan-induced command and step translation)**.** Fix a valid plan P and $t < N$. For each expression occurrence e, let $\widehat{e}_t(\widehat{\rho})$ be its bit-vector translation. A symbolic valuation $\widehat{\rho}$ assigns every variable in Θ_M a bit vector of the width fixed for its declared type, including variables not yet defined by the concrete execution. Write $\widehat{\mathsf{select}}$ and $\widehat{\mathsf{update}}$ for exact encoded array selection and update. The translation is

$$\widehat{z}_t(\widehat{\rho}) = \widehat{\rho}(z),$$
$$\widehat{lit}_t(\widehat{\rho}) = \mathrm{enc}_{\tau_{lit}}(\mathsf{val}(lit)),$$
$$\widehat{e_1[e_2]}_t(\widehat{\rho}) = \widehat{\mathsf{select}}\big(\widehat{e}_{1t}(\widehat{\rho}), \widehat{e}_{2t}(\widehat{\rho})\big),$$
$$\widehat{p(e_1,\dots,e_m)}_t(\widehat{\rho}) = \widehat{p}_{t,\ell}\big(\widehat{e}_{1t}(\widehat{\rho}), \dots, \widehat{e_m}_t(\widehat{\rho})\big),$$

where ℓ is the primitive occurrence. If every value read by e is defined in ρ and $\widehat{\rho}$ encodes those defined components, then

$$\widehat{e}_t(\widehat{\rho}) = \mathrm{enc}_{\tau_e}([\![e]\!]_\rho).$$

For each command occurrence c' in the complete body c, define

$$\mathrm{SymC}_t(c'; \widehat{\rho}, \pi) = (\widehat{\kappa}, \widehat{\rho}', \widehat{\varepsilon}, \mathbf{h}).$$

Here π states that c' is reached, and $\widehat{\kappa}$ states that it is reached and completes normally. The failure component is used only when $\pi \wedge \neg\widehat{\kappa}$ holds. Let $\star$ be an arbitrary fixed vector of the required width, let $\mathbf{0}$ be the all-false hit vector, and let $\mathbf{h}_{\{\gamma\}}$ be the one-hot vector for γ. Boolean operations and ite act componentwise on valuations and hit vectors. The primitive commands translate as

$$\mathrm{SymC}_t(\mathsf{skip}; \widehat{\rho}, \pi) = (\pi, \widehat{\rho}, \star, \mathbf{0}),$$
$$\mathrm{SymC}_t(z := e; \widehat{\rho}, \pi) = \Big(\pi, \widehat{\rho}\big[z \mapsto \mathrm{ite}(\pi, \widehat{e}_t(\widehat{\rho}), \widehat{\rho}(z))\big], \star, \mathbf{0}\Big),$$
$$\mathrm{SymC}_t(z[e_1] := e_2; \widehat{\rho}, \pi) = \Big(\pi, \widehat{\rho}\big[z \mapsto \mathrm{ite}\big(\pi, \widehat{\mathsf{update}}(\widehat{\rho}(z), \widehat{e}_{1t}(\widehat{\rho}), \widehat{e}_{2t}(\widehat{\rho})), \widehat{\rho}(z)\big)\big], \star, \mathbf{0}\Big),$$
$$\mathrm{SymC}_t(\mathsf{mark}(\gamma); \widehat{\rho}, \pi) = (\pi, \widehat{\rho}, \star, \pi \wedge \mathbf{h}_{\{\gamma\}}),$$
$$\mathrm{SymC}_t(\mathsf{fail}(\varepsilon, \gamma_s); \widehat{\rho}, \pi) = (\mathsf{false}, \widehat{\rho}, \mathrm{enc}_{t+1}^X(\varepsilon), \pi \wedge \mathbf{h}_{\{\gamma_s\}}).$$

For sequence, let

$$(\kappa_1, \widehat{\rho}_1, \widehat{\varepsilon}_1, \mathbf{h}_1) = \mathrm{SymC}_t(c_1; \widehat{\rho}, \pi),$$
$$(\kappa_2, \widehat{\rho}_2, \widehat{\varepsilon}_2, \mathbf{h}_2) = \mathrm{SymC}_t(c_2; \widehat{\rho}_1, \kappa_1).$$

Then

$$\mathrm{SymC}_t(c_1; c_2; \widehat{\rho}, \pi) = \big(\kappa_2, \widehat{\rho}_2, \mathrm{ite}(\kappa_1, \widehat{\varepsilon}_2, \widehat{\varepsilon}_1), \mathbf{h}_1 \vee \mathbf{h}_2\big).$$

For a conditional, put $g = \widehat{e}_t(\widehat{\rho})$ and

$$(\kappa_T, \widehat{\rho}_T, \widehat{\varepsilon}_T, \mathbf{h}_T) = \mathrm{SymC}_t(c_T; \widehat{\rho}, \pi \wedge g),$$
$$(\kappa_F, \widehat{\rho}_F, \widehat{\varepsilon}_F, \mathbf{h}_F) = \mathrm{SymC}_t(c_F; \widehat{\rho}, \pi \wedge \neg g).$$

The conditional translation is

$$\mathrm{SymC}_t(\mathsf{if}\ e\ \mathsf{then}\ c_T\ \mathsf{else}\ c_F; \widehat{\rho}, \pi) = \big(\mathrm{ite}(g, \kappa_T, \kappa_F), \mathrm{ite}(g, \widehat{\rho}_T, \widehat{\rho}_F), \mathrm{ite}(g, \widehat{\varepsilon}_T, \widehat{\varepsilon}_F), \mathbf{h}_T \vee \mathbf{h}_F\big).$$

For a loop occurrence ℓ, let m be the nonnegative trip count, τ_m its type, and $\widehat{m}$ its exact encoded value at loop entry. Define the finite comparison predicate

$$\widehat{\mathsf{lt}}_{t,\ell}(k, \widehat{m}) := \bigvee_{\substack{n \in \mathcal{R}_{t,\ell}(m) \\ k < n}} \big(\widehat{m} = \mathrm{enc}_{\tau_m}(n)\big),$$

where an empty disjunction is false, and set

$$L_{t,\ell} = \max\big(\{0\} \cup \mathcal{R}_{t,\ell}(m)\big).$$

Initialize $a_0 = \pi$, $\widehat{\rho}_0 = \widehat{\rho}$, $\widehat{\varepsilon}_0 = \star$, and $\mathbf{h}_0 = \mathbf{0}$. For $k < L_{t,\ell}$, define

$$b_k = a_k \wedge \widehat{\mathsf{lt}}_{t,\ell}(k, \widehat{m}),$$
$$(\kappa_k, \widehat{\rho}_{k+1}, \widehat{\varepsilon}'_k, \mathbf{h}'_k) = \mathrm{SymC}_t(c[j := k]; \widehat{\rho}_k, b_k),$$
$$a_{k+1} = a_k \wedge \big(\neg \widehat{\mathsf{lt}}_{t,\ell}(k, \widehat{m}) \vee \kappa_k\big),$$
$$\widehat{\varepsilon}_{k+1} = \mathrm{ite}(b_k \wedge \neg\kappa_k, \widehat{\varepsilon}'_k, \widehat{\varepsilon}_k), \qquad \mathbf{h}_{k+1} = \mathbf{h}_k \vee \mathbf{h}'_k.$$

The loop translation is

$$\mathrm{SymC}_t(\mathsf{for}\ j < e\ \mathsf{do}\ c; \widehat{\rho}, \pi) = (a_{L_{t,\ell}}, \widehat{\rho}_{L_{t,\ell}}, \widehat{\varepsilon}_{L_{t,\ell}}, \mathbf{h}_{L_{t,\ell}}).$$

It emits no copies for an empty range, executes exactly the first m copies when the loop is reached, and disables every later copy after a failure. At the step boundary, let

$$\widehat{x}_t = \mathrm{enc}^X_t(s_t), \qquad \widehat{u}_t = \mathrm{enc}^U_t(u_t), \qquad \rho_{\mathrm{in},t} = s_t \cup u_t \quad (s_t \in S_t,\ u_t \in U_M).$$

For each local or output variable z, fix an arbitrary bit vector $\widehat{\perp}_z$ of that width. The total symbolic input valuation is

$$\widehat{\rho}_{\mathrm{in},t}(z) = \begin{cases} \mathrm{enc}_{\Theta_S(z)}(s_t(z)) & z \in \mathrm{dom}(\Theta_S), \\ \mathrm{enc}_{\Theta_I(z)}(u_t(z)) & z \in \mathrm{dom}(\Theta_I), \\ \widehat{\perp}_z & z \in \mathrm{dom}(\Theta_L \cup \Theta_O). \end{cases}$$

Only active reads must encode defined concrete values. The fixed vectors make inactive symbolic branches well defined and do not affect an active result. For $t < N - 1$, normal execution determines the next input valuation by

$$\langle c, \rho_{\mathrm{in},t}\rangle \Downarrow \langle \rho_{f,t}, H_t\rangle \quad \Longrightarrow \quad \rho_{\mathrm{in},t+1} = \big(\rho_{f,t}|_{\Theta_S}\big) \cup u_{t+1}.$$

Translating the complete body gives

$$(\widehat{\kappa}_t, \widehat{\rho}_{f,t}, \widehat{\varepsilon}_t, \mathbf{h}_t) = \mathrm{SymC}_t(c; \widehat{\rho}_{\mathrm{in},t}, \mathsf{true}).$$

The symbolic step projects the final valuation onto its state and output components and repacks them:

$$\mathrm{SymStep}_{M,t}(\widehat{x}_t, \widehat{u}_t) = (\widehat{x}_{t+1}, \widehat{y}_t, \mathbf{h}_t),$$
$$(\widehat{x}_{t+1}, \widehat{y}_t, \mathbf{h}_t) = \Big(\mathrm{ite}\big(\widehat{\kappa}_t, \mathrm{Pack}^X_{t+1}(\widehat{\rho}_{f,t}|_{\Theta_S}), \widehat{\varepsilon}_t\big), \mathrm{ite}\big(\widehat{\kappa}_t, \mathrm{Pack}^Y_t(\widehat{\rho}_{f,t}|_{\Theta_O}), \mathrm{enc}^Y_t(\perp_Y)\big), \mathbf{h}_t\Big).$$

Thus

$$\mathrm{SymStep}_{M,t} : \mathbb{B}^{d^X_t} \times \mathbb{B}^{d^U_t} \longrightarrow \mathbb{B}^{d^X_{t+1}} \times \mathbb{B}^{d^Y_t} \times \mathsf{Bool}^{\Gamma_M}.$$

Lemma 5 (Symbolic-step correctness). *For $s \in S_t$ and $u \in U_M$, $\mathrm{SymStep}_{M,t}$ exactly encodes $\bar{\delta}_M$ at layer t. Namely, if $\bar{\delta}_M(s,u) = (s', y, H)$, then*

$$\mathrm{SymStep}_{M,t}\big(\mathrm{enc}_t^X(s), \mathrm{enc}_t^U(u)\big) = \big(\mathrm{enc}_{t+1}^X(s'), \mathrm{enc}_t^Y(y), \mathbf{h}_H\big),$$

where $(\mathbf{h}_H)_\gamma = \mathsf{true}$ exactly when $\gamma \in H$.

Proof. Let $\rho_{\mathrm{in}} = s \cup u$. We prove a stronger claim by structural induction on each command occurrence c'. Fix any path condition π and any assignment under which $\widehat{\rho}$ encodes the defined components of ρ. If

$$(\widehat{\kappa}, \widehat{\rho}', \widehat{\varepsilon}, \mathbf{h}) = \mathrm{SymC}_t(c'; \widehat{\rho}, \pi),$$

then

$$\begin{aligned}
&\pi = \mathsf{false} && \Longrightarrow \widehat{\kappa} = \mathsf{false}, \quad \widehat{\rho}' = \widehat{\rho}, \quad \mathbf{h} = \mathbf{0},\\
&\pi = \mathsf{true}, \quad \langle c', \rho\rangle \Downarrow \langle \rho'', H\rangle && \Longrightarrow \widehat{\kappa} = \mathsf{true}, \quad \widehat{\rho}' = \widehat{\rho''}, \quad \mathbf{h} = \mathbf{h}_H,\\
&\pi = \mathsf{true}, \quad \langle c', \rho\rangle \Downarrow \mathsf{Err}(\varepsilon, H) && \Longrightarrow \widehat{\kappa} = \mathsf{false}, \quad \widehat{\varepsilon} = \mathrm{enc}_{t+1}^X(\varepsilon), \quad \mathbf{h} = \mathbf{h}_H.
\end{aligned}$$

Only defined valuation components are compared. Internal-range containment makes every reached value encodable, and primitive exactness makes every translated expression equal to its concrete value. The displayed primitive translations give the base cases. Sequence activates its second command exactly after normal completion of the first and unions their hits. Exact guard evaluation activates the same conditional arm as the concrete execution. For a reached loop, range containment gives $m \le L_{t,\ell}$. Induction on k shows that a_k holds exactly while the first $\min(k,m)$ copies have completed normally, with $\widehat{\rho}_k$ and $\mathbf{h}_k$ encoding their valuation and accumulated hits. A failing copy records its error and disables every later copy. Hence the loop agrees with $c^{[m]}$. This proves the stronger claim. Apply it to the complete body with $\pi = \mathsf{true}$ and $\rho = \rho_{\mathrm{in}}$. Normal execution gives

$$\bar{\delta}_M(s,u) = \big(\rho'|_{\Theta_S}, \rho'|_{\Theta_O}, H\big).$$

Failure with $\mathsf{Err}(\varepsilon, H)$ gives

$$\bar{\delta}_M(s,u) = (\varepsilon, \bot_Y, H).$$

In the first case, the two packing identities turn the state and output projections of $\widehat{\rho'}$ into the encodings of s' and y. In the second, the failure component is already $\mathrm{enc}_{t+1}^X(s')$, and the output branch selects $\mathrm{enc}_t^Y(\bot_Y)$. The two implications therefore yield the claimed equality. □

Encoded transition. The symbolic step starts from a normal state and may return either a normal or terminal successor. To also encode a step whose current state is already terminal, define

$$\begin{aligned}
&\widehat{\delta}_t^D(\widehat{x}_t, \widehat{u}_t, \widehat{x}_{t+1}, \widehat{y}_t, \mathbf{h}_t)\\
&\quad\equiv \Big[\Big(\bigvee_{s\in S_t} \widehat{x}_t = \mathrm{enc}_t^X(s)\Big) \wedge (\widehat{x}_{t+1}, \widehat{y}_t, \mathbf{h}_t) = \mathrm{SymStep}_{M,t}(\widehat{x}_t, \widehat{u}_t)\Big]\\
&\qquad\vee \bigvee_{\varepsilon\in Err_M} \Big[\widehat{x}_t = \mathrm{enc}_t^X(\varepsilon) \wedge \widehat{x}_{t+1} = \mathrm{enc}_{t+1}^X(\varepsilon)\\
&\qquad\qquad \wedge \widehat{y}_t = \mathrm{enc}_t^Y(\bot_Y) \wedge \bigwedge_{\gamma\in\Gamma_M} \neg h_{t,\gamma}\Big].
\end{aligned}$$

The first disjunct handles normal execution. The second re-encodes the same terminal error at layer $t+1$, emits $\bot_Y$, and records no new hits.

Theorem 5 (Exact Behavior IR encoding). *Let P be a valid Behavior representation plan for (M, D), and let $\mathcal{A}_D$ be the finite layered system induced by P in Lemma 4. The boundary encoding maps $(\mathrm{enc}_t^X)_{t\le N}$, $(\mathrm{enc}_t^U)_{t<N}$, and $(\mathrm{enc}_t^Y)_{t<N}$, together with their inverse maps, validity predicates, $\widehat{I}_D$, $\widehat{K}_N$, and $(\widehat{\delta}_t^D)_{t<N}$, form an exact bounded encoding of $\mathcal{A}_D$ as defined in Definition 15.*

Proof. The component encodings are injective by validity of P. Fixed-order concatenation and the disjoint tags therefore make every boundary map injective. Each inverse is well defined on its image, which the corresponding validity predicate recognizes exactly. The finite enumerations defining $\widehat{I}_D$

and $\widehat{K}_N$ accept exactly the encodings of states in I_D and sequences satisfying K_N, respectively. Fix $t < N$ and a valid encoded tuple $(\widehat{x}, \widehat{u}, \widehat{x}', \widehat{y}, \mathbf{h})$. Decode its first four components as s, u, s', y, and let $H = \{\gamma \in \Gamma_M \mid h_\gamma = \mathsf{true}\}$. The transition of $\mathcal{A}_D$ at layer t is $\bar{\delta}_M$ restricted to $X_t^D \times U_M$. It remains to show that $\widehat{\delta}_t^D$ accepts exactly its encoded transitions:

$$\widehat{\delta}_t^D(\widehat{x}, \widehat{u}, \widehat{x}', \widehat{y}, \mathbf{h}) \iff \bar{\delta}_M(s, u) = (s', y, H).$$

Because $X_t^D = S_t \cup Err_M$, the decoded state s lies in exactly one of these two sets. If $s \in S_t$, only the first disjunct can hold. Lemma 5 states that $\mathrm{SymStep}_{M,t}$ returns the encoding of $\bar{\delta}_M(s, u)$. The state and output encodings are injective, and $\mathbf{h}_H$ identifies H componentwise. Hence this encoded result equals $(\widehat{x}', \widehat{y}, \mathbf{h})$ exactly when $\bar{\delta}_M(s, u) = (s', y, H)$. If $s = \varepsilon \in Err_M$, only the second disjunct can hold, and it is equivalent to $(s', y, H) = (\varepsilon, \bot_Y, \emptyset)$. This is exactly $\bar{\delta}_M(\varepsilon, u)$ by absorption. Hence the encoded transition relation satisfies the remaining condition of Definition 15. □

B.9 Behavior Frontend Fidelity and Audits

Overview. Subsections B.7 and B.8 define the Behavior IR and its exact bounded encoding. This subsection connects the source execution of $\hat{B}$ to that IR through assumptions on step semantics, trace admissibility, the goal catalogue and replay.

Definition 27 (Source-to-IR correspondence)**.** Let $\hat{B}$ be an admitted Behavior model. Its source execution has state, input, output and error sets $X_{\hat{B}}, U_{\hat{B}}, Y_{\hat{B}}$ and $Err_{\hat{B}}$, nonempty initial set $\emptyset \neq I^0_{\hat{B}} \subseteq X_{\hat{B}}$, and the goal catalogue $\Gamma_{\hat{B}}$ of Definition 6. Its fallible step is

$$\delta^0_{\hat{B}} : X_{\hat{B}} \times U_{\hat{B}} \longrightarrow \big(X_{\hat{B}} \times Y_{\hat{B}} \times 2^{\Gamma_{\hat{B}}}\big) \cup \{\mathsf{Err}(\varepsilon, H) \mid \varepsilon \in Err_{\hat{B}},\ H \subseteq \Gamma_{\hat{B}}\}.$$

For a stimulus with $|\sigma| = N$, the harness constructs $u_t^{\hat{B}} \in U_{\hat{B}}$ for $t < N$ from $(\sigma_t, a_t^{\hat{B}}(\xi))$. Thus the source input includes both the accepted transaction batch and the schedule events visible to the Behavior semantics. Let M be the Behavior IR procedure obtained by lowering $\hat{B}$, as in Definition 23. Admission requires M to be well formed and lowering to supply the source-to-IR maps

$$\mathrm{low}_X : X_{\hat{B}} \to X_M, \qquad \mathrm{low}_U : U_{\hat{B}} \to U_M, \qquad \mathrm{low}_Y : Y_{\hat{B}} \to Y_M,$$

and

$$\mathrm{low}_E : Err_{\hat{B}} \to Err_M, \qquad \mathrm{low}_\Gamma : \Gamma_{\hat{B}} \to \Gamma_M.$$

The first four maps relate source values to IR values, while low_Γ maps source goals to their corresponding IR goals. The enc maps of Definition 15 then encode the IR values as solver bit vectors. Extend low_Γ pointwise to goal sets. The component maps induce the following map low on step results:

$$\mathrm{low}(x', y, H) = (\mathrm{low}_X(x'), \mathrm{low}_Y(y), \mathrm{low}_\Gamma(H)),$$
$$\mathrm{low}\big(\mathsf{Err}(\varepsilon, H)\big) = \mathsf{Err}\big(\mathrm{low}_E(\varepsilon), \mathrm{low}_\Gamma(H)\big).$$

Let $\bot_{\hat{B}}$ be the padding output of the source absorption. Extend low_X to terminal errors using low_E, set $\mathrm{low}_Y(\bot_{\hat{B}}) = \bot_Y$, and apply low componentwise to absorbed traces. For a set W of source traces, write $\mathrm{low}(W) = \{\mathrm{low}(w) \mid w \in W\}$. For the IR domain D fixed in Subsection B.8, write

$$D_M := D = \big(\mathcal{M}_M(I^0_M), N, K_N\big).$$

Let $K_N^{\hat{B}}$ be the source trace predicate and set

$$D^{\mathrm{src}}_{\hat{B}} = \big(\mathrm{Abs}_{I^0_{\hat{B}}}(\delta^0_{\hat{B}}), N, K_N^{\hat{B}}\big).$$

Assumption 5 (Behavior semantic fidelity)**.** For every $s \in X_{\hat{B}}$ and $u \in U_{\hat{B}}$,

$$\delta^0_M\big(\mathrm{low}_X(s), \mathrm{low}_U(u)\big) = \mathrm{low}\big(\delta^0_{\hat{B}}(s, u)\big).$$

Thus lowering preserves the normal successor and output, or the failure, together with the emitted goal set. The absorbed extensions in Definition 27 preserve the same correspondence after a failure.

Assumption 6 (Behavior catalogue fidelity)**.** The goal policy and lowering induce a bijection

$$\mathrm{low}_\Gamma : \Gamma_{\hat{B}} \xrightarrow{\sim} \Gamma_M,$$

so every source goal has exactly one corresponding IR goal and every IR goal arises in this way. A source goal γ is hit on a source step exactly when $\mathrm{low}_\Gamma(\gamma)$ is hit on the corresponding IR step. The bijection preserves the goal classes:

$$\mathrm{low}_\Gamma(\Gamma^{\mathrm{cov}}_{\hat{B}}) = \Gamma^{\mathrm{cov}}_M, \qquad \mathrm{low}_\Gamma(\Gamma^{\mathrm{safe}}_{\hat{B}}) = \Gamma^{\mathrm{safe}}_M.$$

Assumption 7 (Behavior domain and replay fidelity)**.** The lowering maps preserve the initial states and inputs:

$$\mathrm{low}_X(I^0_{\hat{B}}) = I^0_M, \qquad \mathrm{low}_U(U_{\hat{B}}) = U_M.$$

Thus every IR initial state and input has at least one source representative. For every source state, input and output sequence of lengths $N+1$, N and N, respectively,

$$K^{\hat{B}}_N(\mathbf{x}, \mathbf{u}, \mathbf{y}) \iff K_N\big(\mathrm{low}_X(\mathbf{x}), \mathrm{low}_U(\mathbf{u}), \mathrm{low}_Y(\mathbf{y})\big),$$

where

$$\mathrm{low}_X(\mathbf{x}) = \big(\mathrm{low}_X(x_t)\big)_{t\le N}, \qquad \mathrm{low}_U(\mathbf{u}) = \big(\mathrm{low}_U(u_t)\big)_{t<N}, \qquad \mathrm{low}_Y(\mathbf{y}) = \big(\mathrm{low}_Y(y_t)\big)_{t<N}.$$

Every raw record returned by $\textsc{Run}_{D_M}(x_0, \mathbf{u})$ carries $s_0 \in I^0_{\hat{B}}$ and $\mathbf{v} \in U^N_{\hat{B}}$ such that

$$\mathrm{low}_X(s_0) = x_0, \qquad \mathrm{low}_U(v_t) = u_t \quad (t < N).$$

It records the complete source replay from $(s_0, \mathbf{v})$, and the IR trace in the record is obtained by applying low to that replay.

Corollary 2 (Behavior trace and goal correspondence)**.** *Under Assumptions 5, 6 and 7,*

$$\mathrm{low}\big(\mathrm{Tr}(D^{\mathrm{src}}_{\hat{B}})\big) = \mathrm{Tr}(D_M), \qquad \mathrm{low}_\Gamma\big(\mathrm{Reach}_{D^{\mathrm{src}}_{\hat{B}}}\big) = \mathrm{Reach}_{D_M}.$$

If P is a valid Behavior representation plan for (M, D_M), the hypotheses of Theorem 3 hold for D_M, and Algorithm 3 returns (Rch_M, Unr_M, Unk_M, W) for $G_M \subseteq \Gamma_M$. Define

$$G_{\hat{B}} = \mathrm{low}^{-1}_\Gamma(G_M),$$
$$(Rch_{\hat{B}}, Unr_{\hat{B}}, Unk_{\hat{B}}) = \big(\mathrm{low}^{-1}_\Gamma(Rch_M), \mathrm{low}^{-1}_\Gamma(Unr_M), \mathrm{low}^{-1}_\Gamma(Unk_M)\big).$$

These three sets partition $G_{\hat{B}}$, with

$$Rch_{\hat{B}} \subseteq \mathrm{Reach}_{D^{\mathrm{src}}_{\hat{B}}}, \qquad Unr_{\hat{B}} \cap \mathrm{Reach}_{D^{\mathrm{src}}_{\hat{B}}} = \emptyset,$$

and every goal in $Rch_{\hat{B}}$ has a source replay witness.

Proof. Step induction using Assumptions 5 and 7 gives

$$w_{\hat{B}} \in \mathrm{Tr}(D^{\mathrm{src}}_{\hat{B}}) \Longrightarrow \mathrm{low}(w_{\hat{B}}) \in \mathrm{Tr}(D_M),$$
$$w_M \in \mathrm{Tr}(D_M) \Longrightarrow \exists w_{\hat{B}} \in \mathrm{Tr}(D^{\mathrm{src}}_{\hat{B}}) : \mathrm{low}(w_{\hat{B}}) = w_M.$$

The reverse implication chooses the source representatives guaranteed by Assumption 7. Hence the trace sets are equal after applying low. Hit preservation and the catalogue bijection give, for every $\gamma \in \Gamma_{\hat{B}}$,

$$\gamma \in \mathrm{Reach}_{D^{\mathrm{src}}_{\hat{B}}} \iff \mathrm{low}_\Gamma(\gamma) \in \mathrm{Reach}_{D_M}.$$

A valid P supplies the exact instance used by Theorem 3. Applying low^{-1}_Γ to that theorem gives the stated classification. The source replays carried by the audited records provide the witnesses. □

Catalogue and replay audits. Before any query, the catalogue audit checks the goal bijection, classes and hit correspondence. For each raw record, the replay audit runs its source representatives, checks that the resulting source and IR traces correspond and satisfy their trace predicates, and applies Definition 17 with $\widehat{\delta}^{D_M}_t$. Only audited traces are retained.

B.10 Artifact Instances and Reported Bounded Conclusions

Overview. The preceding subsections construct exact bounded instances for Behavior and RTL. This subsection pairs each instance with the audited traces from concrete evaluation and states when the common analysis may report a goal as reached, bounded-unreachable or unresolved.

Analysis domains. Behavior is analyzed through its lowered IR domain D_M, whereas RTL is analyzed directly in $D_{\hat{R}}$. Accordingly, for $A \in \{\hat{B}, \hat{R}\}$, write $D_A = D_M$ when $A = \hat{B}$, and $D_A = D_{\hat{R}}$ when $A = \hat{R}$. When the required exact instance can be constructed, a valid Behavior representation plan and Theorem 5 supply the instance for D_M, while Theorem 4 supplies the instance

for $D_{\hat{R}}$. The exact Behavior instance classifies Γ_M, and its statuses are mapped back to $\Gamma_{\hat{B}}$ through low_Γ^{-1}. The exact RTL instance classifies $\Gamma_{\hat{R}}$ directly. An RTL reset restoration claim is reported only when its scenario goal lies in Rch and its violation goal lies in Unr.

Evaluated pool. Apply $\textsc{Audit}_{D_A}$ to every record in $\mathcal{E}_A$. A Refused or Inconsistent result stops analysis, while an Outside record is discarded. Otherwise the retained records form $W^A_{\mathrm{eval}} \subseteq \mathrm{Tr}(D_A)$. The Behavior audit reconstructs traces in D_M, while the RTL audit reconstructs traces in $D_{\hat{R}}$. A nonempty W^A_{eval} is the initial pool W_0. If it is empty, the initial-pool procedure of Subsection B.3 checks feasibility. Later queries may add audited witnesses to W, but they do not change Ξ_A, $\mathcal{E}_A$, or the output score.

Corollary 3 (Reported bounded conclusions)**.** *For either artifact, set*

$$(\Gamma, \Gamma^{\mathrm{cov}}, \Gamma^{\mathrm{safe}}) = \begin{cases} (\Gamma_M, \Gamma_M^{\mathrm{cov}}, \Gamma_M^{\mathrm{safe}}) & \textit{for Behavior}, \\ (\Gamma_{\hat{R}}, \Gamma_{\hat{R}}^{\mathrm{cov}}, \Gamma_{\hat{R}}^{\mathrm{safe}}) & \textit{for RTL}. \end{cases}$$

Under the hypotheses of Theorem 3 for D_A, suppose Algorithm 3 classifies Γ and returns

$$(Rch,\, Unr,\, Unk, W).$$

For $C \in \{\Gamma^{\mathrm{cov}}, \Gamma^{\mathrm{safe}}\}$,

$$Unk \cap C = \emptyset \quad \Longrightarrow \quad \begin{array}{l} Rch \cap C = \mathrm{Reach}_{D_A} \cap C, \\ Unr \cap C = C \setminus \mathrm{Reach}_{D_A}. \end{array}$$

For $C = \Gamma^{\mathrm{cov}}$, the premise $Unk \cap C = \emptyset$ establishes bounded coverage closure*. For $C = \Gamma^{\mathrm{safe}}$, if $Unk \cap C = \emptyset$ and $Rch \cap C = \emptyset$, then* $\mathrm{Reach}_{D_A} \cap C = \emptyset$*, establishing* bounded safety*.*

Proof. Since the three status sets partition Γ and $Unk \cap C = \emptyset$,

$$C = (Rch \cap C) \cup (Unr \cap C).$$

Theorem 3 gives $Rch \subseteq \mathrm{Reach}_{D_A}$ and $Unr \cap \mathrm{Reach}_{D_A} = \emptyset$. The first inclusion gives

$$Rch \cap C \subseteq \mathrm{Reach}_{D_A} \cap C.$$

Conversely, let $\gamma \in \mathrm{Reach}_{D_A} \cap C$. The displayed partition places γ in Rch or Unr. The second property excludes Unr, so $\gamma \in Rch \cap C$. This proves the first equality. The partition and that equality then give

$$Unr \cap C = C \setminus (Rch \cap C) = C \setminus \mathrm{Reach}_{D_A}.$$

For $C = \Gamma^{\mathrm{safe}}$, the additional premise $Rch \cap C = \emptyset$ and the first equality give $\mathrm{Reach}_{D_A} \cap C = \emptyset$. □

B.11 Trusted Boundary, Claim Scope and Conclusion

Trusted boundary. The Behavior result depends on Assumptions 5-7 and a valid representation plan. The RTL result depends on Assumptions 2-4 and the exact component encodings required by Theorem 4. Both results rely on correct implementations of the encoded queries, replay harness, catalogue audit and replay audit. They also trust every unsatisfiable solver result used to establish infeasibility or place a goal in Unr. Replay may expose fidelity violations, but successful checks do not establish fidelity for every admitted trace. The task-declared transaction interface, endpoint bindings and selected protocol adapter are trusted to describe the intended hardware interface.

Claim scope. The output score concerns only Ξ_A, while goal classification concerns the fixed bounded domains. RTL is classified in $D_{\hat{R}}$. Behavior is analyzed in D_M, with conclusions transferred to $D^{\mathrm{src}}_{\hat{B}}$ under the assumptions relating the source and IR semantics. The results establish neither Behavior-RTL equivalence nor properties outside the catalogues, and make no claim beyond the fixed horizon and admissibility predicates. Catalogue and predicate adequacy remain modeling choices. The RTL claim covers discrete event slot executions over the admitted finite logic domain under the fixed adapter and environment contract, not analog timing or schedules outside that contract.

Conclusion. The construction provides a common audited bounded analysis for admitted Behavior and RTL artifacts. Under the stated assumptions, every reached goal has an audited witness. A goal classified as bounded-unreachable cannot be hit within the fixed domain. If no coverage goal remains unresolved, bounded coverage closure holds. If every safety goal is classified as bounded-unreachable, bounded safety holds. The analysis leaves artifact scores unchanged.

C Implementation, Experimental Details and Additional Results

Section C.1 describes the agent environment and tools, and Section C.2 gives inference and evaluation settings. Section C.3 examines the effect of goal-guided search on RTL coverage. We detail the Behavior IR-HLS comparison in Section C.4, pipeline comparison and PPA exploration in Section C.5, and training and self-improvement in Section C.6. Section C.7 summarizes resource use and costs. Sections C.8 and C.9 provide a modeling comparison and a self-improvement case study, respectively.

C.1 Agent Environment and Tools

Execution environment. The Linux namespace-based sandbox provides a persistent writable workspace, with the task specification and public documentation available as read-only files. Agents use `run_command` for file operations and shell-based development tests. Depending on the experimental setting, additional tools support simulation comparison, PPA measurement, and artifact submission or search completion. These interfaces are detailed in the corresponding sections.

Development tools. Development uses Python for Behavior models and Verilator (Snyder et al., 2026) with cocotb (cocotb contributors, 2025) for RTL simulation and testing. ChipMATE and MAGE use Icarus Verilog (Williams, 2026) during development. The HLS experiments use AMD Vitis HLS 2025.2 (AMD, 2026) for C++-to-RTL synthesis. PPA exploration uses Yosys/ABC (Wolf & Glaser, 2013; Brayton & Mishchenko, 2010) for synthesis and technology mapping, OpenROAD (Ajayi et al., 2019) for placement and netlist optimization, and OpenSTA (Parallax Software, Inc., 2026) for timing analysis with the SkyWater 130 nm high-density standard-cell library (SkyWater PDK Authors, 2020) and the Nangate 45 nm standard-cell library (Nangate, 2008), which is based on the predictive FreePDK45 process design kit (Stine et al., 2007).

C.2 Inference and Evaluation Settings

Models and inference. Table 6 lists model identifiers, providers and reasoning settings. We serve local models with SGLang (Zheng et al., 2024) on a single H200 node. We retain default temperature and top-p settings where configurable. GPT-6 Astra is used for dataset construction.

Table 6: **Model and inference configurations.**

Model	Model ID	Access	Reasoning
Qwen3.5-4B	`Qwen/Qwen3.5-4B`	SGLang	Thinking enabled
Qwen3.5-9B	`Qwen/Qwen3.5-9B`	SGLang	Thinking enabled
Qwen3.8-27B	`Qwen/Qwen3.8-27B`	SGLang	Extra-high
GPT-5.6 Luna	`gpt-5.6-luna`	OpenAI	Medium (default)
GPT-5.6 Terra	`gpt-5.6-terra`	OpenAI	Medium (default)
GPT-6 Astra	`gpt-6-astra`	OpenAI	Extra-high / High
Claude Sonnet 5	`claude-sonnet-5`	Anthropic	Adaptive
Claude Fable 5.1	`claude-fable-5-1`	Anthropic	Adaptive
Claude Haiku 4.5	`claude-haiku-4-5-20251001`	Anthropic	Thinking enabled
DeepSeek-V4.1-Flash	`deepseek-v4p1-flash`	Fireworks	High (default)

Rollout settings. We run one rollout per task and model, using a 128K-token (131,072-token) context window and at most 60 turns. When the window fills, we discard the oldest interaction turns while retaining the task instructions and latest tool feedback. Workspace files persist throughout the rollout.

Test configuration. We use 50 baseline sequences combining edge-case patterns and pseudorandom inputs, each requesting 100 transactions per input endpoint. Tests run independently from reset and as continuous workloads. Benchmark scoring uses a 1200 s total budget per artifact, including up to 600 s for goal-guided search. A pass requires successful completion of all baseline tests and no detected artifact failure. Goal-guided search may stop at its budget without invalidating completed checks. If no artifact failure is found, an incomplete baseline is unresolved. Unresolved evaluations count as zero in reporting and RL rewards. Stimulus-generation details are given in Appendix A.7.

C.3 Effect of Goal-Guided Search on RTL Coverage

Search coverage. Table 7 compares average per-task RTL coverage-goal hit rates on the same reference implementations. Each search setting extends the same baseline with a 600 s budget, shared between directions for two-sided search. Search improves coverage on all four datasets; two-sided search has the highest overall average.

Table 7: **Cumulative RTL coverage-goal hit rates (%).** Each search setting augments the same baseline. Avg. weights tasks equally; bold marks column maxima.

Configuration	VerilogEval-v2	RTLLM-v2	CVDP-cid003	BEHAVE (Train + Eval)	Avg.
Baseline	91.75	96.61	87.83	94.76	93.48
+ Reference-directed	92.23	**97.61**	88.88	95.24	94.06
+ Artifact-directed	**92.24**	97.10	**89.10**	95.39	94.16
+ Two-sided	**92.24**	97.43	88.39	**95.59**	**94.21**

Example. RTL generated by GPT-5.6 Luna for the CVDP phase-rotation task passes the full baseline (8,200 checked inputs). Artifact-directed search then finds a sequence that exposes an error: the RTL reads the wait parameter `i_wait` after calculation instead of saving it when the operation starts. Native replay confirms the failure, with `o_subsampling`=1 instead of the expected 0. This example shows how search can expose bugs in multi-stage control logic that baseline tests miss.

C.4 Behavior IR and HLS Comparison Settings

Tasks and references. We select 20 tasks across seven domains whose source implementations can be executed directly under a fixed input contract. Both routes receive the same source code, invocation, numerical semantics and input/output bit representations, with a target-specific interface and a shared representation guide. The golden Behavior models are validated against direct executions of the source programs and remain hidden from the agent. Figure 14 summarizes the comparison.

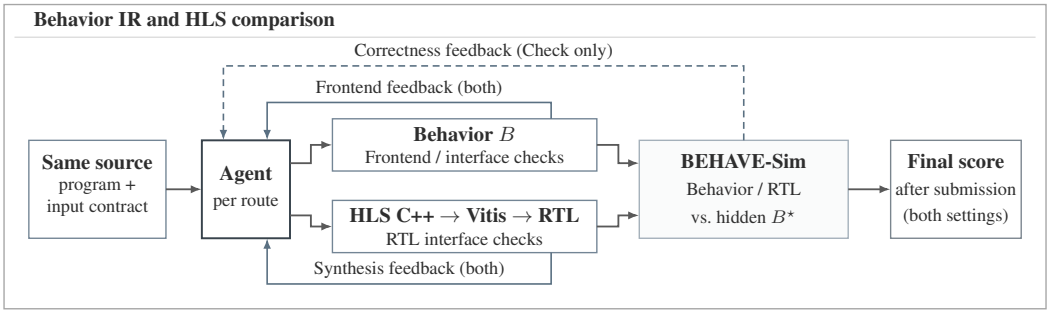


Figure 14: **Behavior IR and HLS comparison workflow.** Both settings provide frontend or synthesis feedback. Only Check returns development-time correctness feedback (dashed arrow). Both settings evaluate the submitted artifact against the same hidden reference.

Comparison settings. We use GPT-5.6 Luna with the rollout and baseline settings in Appendix C.2. Vitis HLS 2025.2 targets `xc7z020clg400-1` at a 25 ns clock period, with a 1200 s synthesis budget. Tool time sums development-check wall times, excluding model generation, queueing and final evaluation. We report the median and interquartile range across tasks.

C.5 Pipeline Comparison and PPA Exploration Settings

Comparison settings. RTL-only, ChipMATE and MAGE receive specifications without the Behavior interface. ChipMATE (Yu et al., 2026b) uses three RTL/Python pairs per round, up to three rounds and 1,000 random inputs per comparison. MAGE (Zhao et al., 2025c) allows four simulation-review

iterations in its no-golden workflow and samples 20 RTL candidates, selecting two for up to 15 editor steps each. Mean API cost includes all candidate-generation and repair calls, averaged over all tasks.

Baseline adaptations. We adapt released implementations to our model and sandbox interfaces, retaining their roles and development tests. For ChipMATE, we remove repair-context clipping and correct output matching and zero-score candidate selection. For MAGE, we fix the no-golden prompt, editor dispatch and compilation/simulation failure handling. Our code includes all patches.

PPA exploration. For MXFP4, both settings jointly develop RTL and Behavior models with area and timing feedback. The Sim-enabled setting additionally compares the agent's RTL and Behavior during development. For each model and setting, we run three searches of at most 60 turns, with initial hints of 1, 4 or 32 parallel processing lanes. These hints guide exploration without constraining the final architecture. Each search explores area-time trade-offs and retains intermediate candidates. For NVFP4, each model runs five searches without development-time BEHAVE-Sim feedback, with initial hints of 1, 2, 4, 8 or 16 lanes.

Measurement and reporting. MXFP4 candidates use a fixed clock $f_0 = 80\,\mathrm{MHz}$ and the same 64-block workload, with 32 bfloat16 (BF16) values per block. Inputs are offered continuously under ready/valid handshaking. Execution time is $\tau_W(R) = N_W(R)/f_0$, where $N_W(R)$ counts cycles from the first input offered to the final output accepted. MXFP4 measurements use the SkyWater 130 nm high-density (HD) standard-cell library (SkyWater PDK Authors, 2020) at the typical-typical process corner (25 °C, 1.80 V). Cell area and setup/hold timing are measured on the same placed and repaired netlist, with interconnect resistance and capacitance estimated from placement. NVFP4 uses the Nangate 45 nm standard-cell library (Nangate, 2008), based on FreePDK45 (Stine et al., 2007), at 400 MHz with a 64-block workload of 16 BF16 values per block. We report nondominated area-time points among candidates that pass independent golden-Behavior and timing checks and complete the full workload.

C.6 Training and Self-Improvement Settings

Task-based RL. We use Slime (Zhu et al., 2025b) with Megatron (Narayanan et al., 2021) for synchronous GRPO (Shao et al., 2024). Hyperparameters are listed in Table 8.

Table 8: Training hyperparameters shared by fixed-dataset RL and self-improvement.

Setting	Value
Model	Qwen3.8-27B
Parameter updates and precision	Full-parameter, BF16
GPUs	8 NVIDIA H200
Training parallelism (tensor/context/pipeline)	4 / 2 / 1
Rollout serving	2 SGLang engines, tensor parallelism 4
Maximum concurrent conversations	32
Update batch	8 tasks $\times$ 8 rollouts
Microbatch size	1
Sampling	Temperature 1, top-p 1, top-k disabled
Repetition/presence/frequency penalties	None
Reasoning	Thinking enabled, `xhigh` effort
Context window	131,072 tokens
Maximum model turns	60
Optimizer	Adam, $(\beta_1, \beta_2) = (0.9, 0.98)$
Learning rate	10^{-6} (constant)
Weight decay	0.1
Gradient-norm clip	1.0
Policy-ratio clip	$[0.8, 1.2]$
KL and entropy loss coefficients	0
Attention and hidden dropout	0
CPU optimizer offload	Enabled
Activation recomputation	Full

Context management. When the training context window fills, we clear interaction history but retain task instructions and workspace files. Evicting individual turns produces overlapping long sequences with high training cost. Within each window, we merge turns with matching token prefixes into one causal sequence to avoid recomputing shared context. Evaluation follows the settings in Section C.2.

Loss normalization. A rollout's segments share one advantage. We normalize each rollout's loss by its loss-bearing token count across segments, then average across rollouts, so segmentation does not increase its weight. Only model-generated tokens contribute, including reasoning and tool calls.

Task pool. We oversample up to 12 concurrent task groups, collecting the first eight complete groups with nonconstant rewards per update. Uniform-reward groups are replaced with new ones. Tasks with all-1 or all-0 rewards enter cooldown for ten or two updates, respectively, then rejoin the pool. Sampling prioritizes tasks without a completed group, then those least recently observed.

Training data and budgets. Fixed-dataset RL samples from the 540-task BEHAVE-Train pool for 40 updates, with evaluation every five updates. Self-improvement starts from a 60-task subset of BEHAVE-Train, selected to match domain proportions while favoring diverse, higher-complexity tasks. After three initial updates, we run 5 task-acquisition rounds with three updates each, for 18 updates in total. Evaluation occurs every three updates. Self-improvement adds 100 tasks to reach a 160-task pool.

Self-improvement step 1: Analysis and acquisition. Every three updates, five parallel Analyst instances run a proposal batch using the current training policy. They receive disjoint sets of complete task groups from the latest rollout, keeping each task's attempts together. Each Analyst targets six proposals within 100 turns, searching a fixed source-only library of 374 files or code excerpts from 67 repositories. Source files used by evaluation tasks and exact content duplicates are excluded. No preconstructed specification-behavior pairs are provided. Experience is consolidated between batches, retained across rounds and visible only to Analysts.

Self-improvement step 2: Construction and admission. Construction proceeds in parallel, with up to 100 turns per role attempt. Each task allows up to three construction-and-review rounds, including the initial draft. Revisions update existing drafts based on feedback. Training resumes at 20 admissions. If the first batch falls short, we allow another batch, then proceed with admitted tasks.

Self-improvement step 3: Model updates. Admitted tasks join the cumulative training pool under the same sampling and cooldown rules. We continue training for three GRPO updates and reuse the latest training rollouts for the next acquisition round.

C.7 Computational Resources and Cost

Resource accounting. Table 9 summarizes GPU-hours and recorded API costs for selected experiments.

Table 9: **Resources and recorded costs of selected experiments.**

Experiment	Scope	Resource use / API cost
Fixed-dataset RL	40 updates	883.5 H200 GPU-hours
Self-improvement	18 updates with task acquisition	537.7 H200 GPU-hours
Evaluation	5 API models $\times$ 381 tasks	\$959.12
Pipeline comparison	5 workflows $\times$ 60 tasks	\$10.54
PPA exploration	12 MXFP4 + 10 NVFP4 searches	\$236.92

C.8 Pipeline Comparison Example

a

```
def eval(self, inputs: dict) -> dict:
  ...
  elif self.state == 1:
    ...
    self.remainder = next_remainder
    self.quotient = next_quotient
    if self.count == 31:
      ...
      self.state = 2
      self.count = 0
    else:
      self.count = (
        (self.count + 1) & 0x3F)
  elif self.state == 2:
    if out_ready:
      self.state = 0
  ...
```

b

```
def process(self, in_n, in_d):
  quotient = 0
  remainder = 0
  if in_d == 0:
    quotient = 4294967295
    remainder = in_n
  else:
    for offset in range(32):
      bit = 31 - offset
      remainder = (
        (remainder << 1) |
        ((in_n >> bit) & 1))
      if remainder >= in_d:
        remainder = remainder - in_d
        quotient = quotient | (1 << bit)
  return {"out_values":
    quotient | (remainder << 32)}
```

Figure 15: **Comparison example.** (a) ChipMATE Python excerpt from a GPT-5.6 Luna rollout. (b) The task's Behavior reference.

Comparison example. Figure 15 shows 32-bit unsigned division. ChipMATE preserves a counter and partial results across clock-level `eval()` calls, whereas the Behavior reference completes the calculation within one `process()` call.

C.9 Additional Results for RL Training and Self-Improvement

Learning dynamics. Figure 16 reports performance across all four benchmarks. Figure 17 shows per-update optimization measurements for fixed-dataset RL and self-improvement.

Case study. Figure 18 follows task acquisition and rollouts on the seed task. The agent turns rollout difficulties into focused practice through task construction. Later progress on the original, more complex task shows that improvement during continued training extends beyond the acquired task.

D Dataset Construction and Validation

This appendix documents the benchmarks used in our experiments. We describe their composition and sources (D.1), BEHAVE construction (D.2), external benchmark adaptation (D.3), human review (D.4), and data splits and release (D.5).

D.1 Dataset Overview and Sources

Dataset composition. BEHAVE covers real-world algorithms and system components across the domains in Table 10. Appendix D.5 describes its training and evaluation splits. Table 11 summarizes behavioral and interface features across the two BEHAVE splits. For evaluation only, we adapt all tasks from VerilogEval-v2 (Pinckney et al., 2025a), RTLLM-v2 (Liu et al., 2024c), and the cid003 subset of CVDP (Pinckney et al., 2025b). Appendix D.3 details these adaptations.

Task sources. We select algorithms and system components from high-level implementations, supplemented by algorithm descriptions where needed. The task source list in our code repository identifies the files or documents used for each task, along with available revisions and license notices. Appendix D.2 describes their adaptation into hardware-oriented tasks.

Task size. Table 12 summarizes the sizes of the golden Behavior models and auxiliary RTL. We count nonblank, non-comment lines in task-local source files, excluding shared libraries.

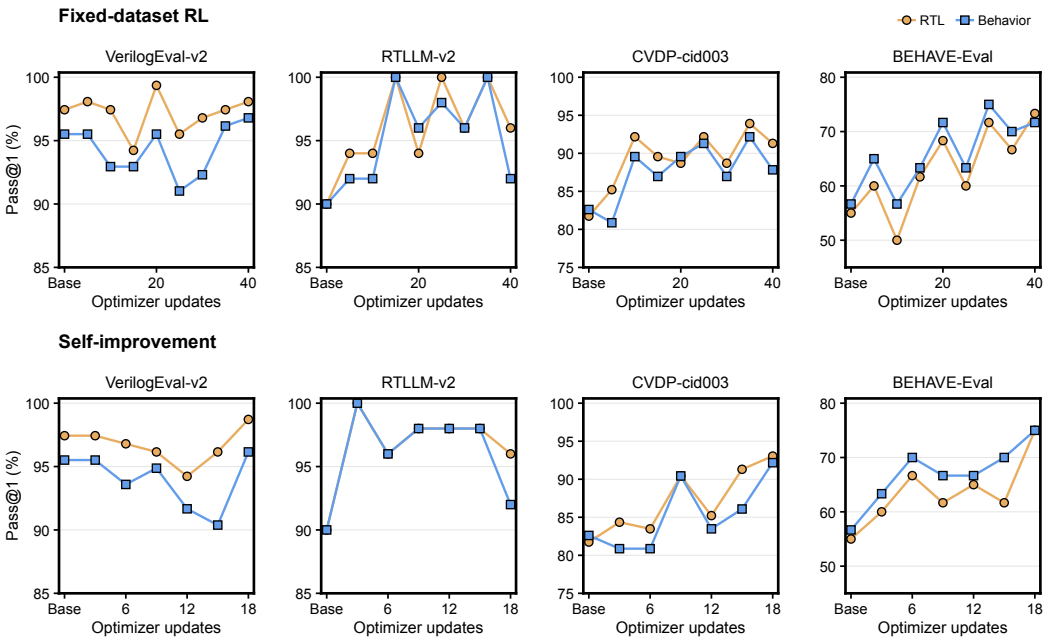


Figure 16: **Checkpoint performance across benchmarks.** Top: fixed-dataset RL; bottom: self-improvement. RTL and Behavior pass@1 are evaluated on the same benchmark tasks at each checkpoint.

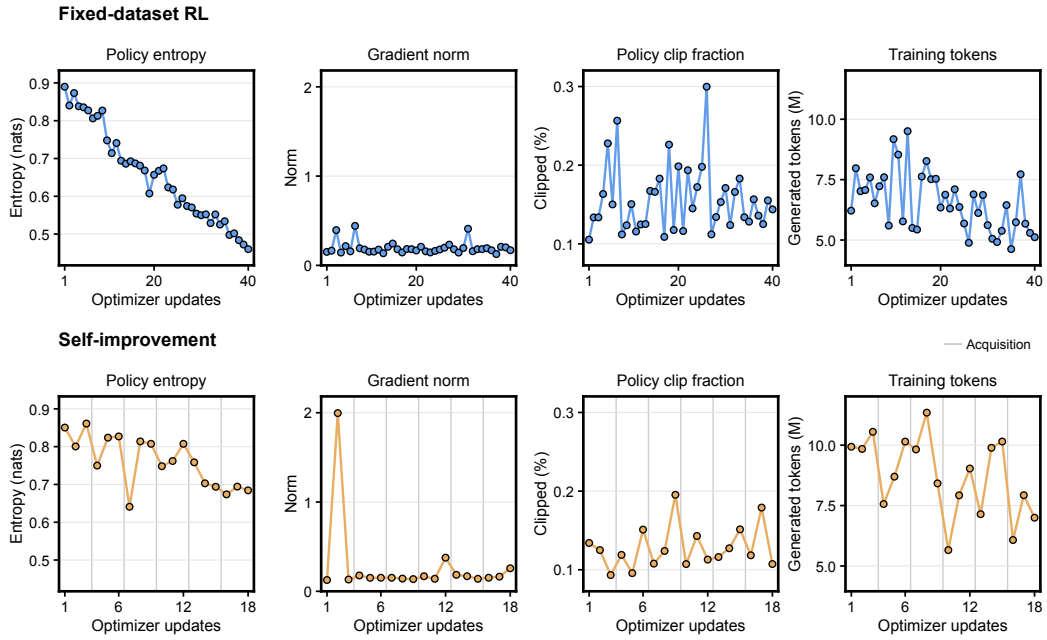


Figure 17: **Training dynamics.** Top: fixed-dataset RL; bottom: self-improvement. Token counts include only generated tokens used for the update. Gray vertical lines mark task acquisitions.

Self-improvement case study: masked top-k attention

Following the workflow in Section 3.4, the agent acquires a related scoring-and-selection task, continues training, and revisits the original attention task.

1. Observed difficulty: masked top-k softmax attention. The seed task combines dot-product scores, key selection, fixed-point exponential tables and normalization. Before update 3, all eight rollouts exhausted the 60-turn budget without submitting. A revisit before update 9 still yielded only 0/8 RTL and 1/8 Behavior passes. Citing these traces, the Analyst proposed isolating integer scoring and masked selection from the full attention computation.

2. Acquisition: integer scoring and masked top-k selection. In the third acquisition round, the Analyst drew on ranked-selection code from entmax. The Builder constructed a task that computes exact integer dot-product scores and returns the selected mask and maximum selected score. The specification fixes lower-index tie breaking, empty masks and keep counts exceeding the eligible set. It retains ranked selection rather than entmax normalization, omitting exponential tables, division and weighted-value accumulation.

3. Verification and admission. The Builder supplied the specification and Behavior reference, and the independent Implementer reconstructed a second Behavior model. Their BEHAVE-Sim comparison passed. After reviewing the source and selection rules, the Reviewer requested one textual revision to remove Behavior-only reset instructions from the hardware specification. The revised task was admitted without a reference RTL implementation.

4. Training and revisiting the seed. The acquired task first entered training at update 10: six rollouts passed both artifacts and two exhausted the turn budget, yielding six rewards of 1 and two of 0. Before update 17, the original attention task reached 2/8 RTL and 3/8 Behavior passes, including two attempts with both artifacts correct. The acquired selection task subsequently reached 8/8 joint passes before update 18; this constant-reward group was excluded from training.

Figure 18: **A recorded self-improvement case.** After acquiring and training on a related task, the agent produces correct RTL and Behavior solutions on the original seed task.

Table 10: **Overview of the 600 source-derived tasks.** Representative task families and sources are shown for each domain. The complete task source list is provided in our code repository.

Domain	Tasks	Representative families	Representative sources
AI/ML	190	Attention, normalization, quantization	Transformers, TorchAO
Numerics	160	Linear algebra, reductions, interpolation	CMSIS-DSP, LAPACK
Signal processing	80	Filtering, transforms, resampling, modulation	liquid-dsp, librosa, Opus
Cryptography	80	Symmetric primitives, field arithmetic, post-quantum kernels	PyCryptodome, liboqs, HEXL
Control	30	Coordinate transforms, controllers, estimation	SimpleFOC, FilterPy
Compression	30	Entropy coding, stream packing, match operations	FLAC, Brotli, Zstd
Data systems	30	Vector search, hashing, bitmap operations	Faiss, hnswlib, CRoaring
Total	**600**		

D.2 Source-Based Task Construction

Human-agent collaboration. Researchers defined application domains, guided source selection and task scope, reviewed and revised tasks. The agent developed specifications and Behavior models, revising them using execution results and human feedback.

Construction workflow. Builder jointly constructed S and B_1 from the sources, specifying numerical and interface rules. In a separate context, Implementer produced B_2 using only S and

Table 11: **Behavioral and interface features of BEHAVE.** Features overlap. A 1:1 mapping denotes one output transaction per input transaction and does not imply low implementation complexity.

Feature	Representative behavior	Train	Eval	Total
Single-stream 1:1 input/output	Block quantization	492	56	548
Single-stream non-1:1 input/output	Filtering, accumulation, interpolation	47	4	51
Multiple independent streams	Three-input, two-output crossbar	1	0	1
External memory access	Memory-backed vectors, lookup tables	26	2	28
External memory writes	In-place updates, writeback	8	1	9

Table 12: **Specification and code size by task collection.** Sizes are medians [interquartile ranges]. Spec words count whitespace-separated units, including interface declarations. LOC: lines of code.

Collection	Tasks	Spec words	Behavior LOC	RTL LOC
VerilogEval-v2	156	270.5 [211.5, 327.5]	9 [7, 13]	11.5 [7, 21]
RTLLM-v2	50	312 [231.75, 427]	14 [9, 24]	20.5 [15, 33.75]
CVDP-cid003	115	473 [358.5, 615.5]	28 [16.5, 47]	34 [21, 53]
BEHAVE (ours)	**600**	**631.5 [496.75, 861.5]**	**40 [24, 69]**	**97 [56, 183]**
Train	540	621 [498.75, 869.75]	39 [24, 67.25]	97 [56, 177.5]
Eval	60	645.5 [481.25, 808]	43.5 [22, 70.25]	102.5 [60.75, 209.5]

public interface documentation. Reviewer checked S against the sources and both models against S, requesting revisions as needed. Material changes to S triggered a fresh reconstruction of B_2. This follows the workflow in Section 3.4. All tasks then underwent human review before acceptance. Researchers made corrections directly or requested revisions from the agent, then approved each task's final version. See Appendix D.4 for review details.

Auxiliary RTL and tools. We also asked the agent to generate reference RTL for each task to support benchmark reuse. This auxiliary artifact is not required for evaluation, which uses the Behavior reference. Both the agent and researchers could use BEHAVE-Sim throughout construction to test and compare implementations. Executable agreement is insufficient because implementations may share a misunderstanding. Human review therefore checked that the NL specification clearly stated the intended behavior and that the Behavior model faithfully implemented it.

D.3 External Benchmark Adaptation

Framework integration. Our goal was to evaluate candidate implementations across all datasets using the same BEHAVE-Sim procedure. We therefore integrated the external benchmarks into our framework rather than using their original testbenches as separate evaluators. For each task, we paired an NL specification with an executable Behavior reference and provided auxiliary reference RTL.

Source issues. We also identified inconsistencies and omissions in the original materials used for adaptation. For example, in the RTLLM-v2 `edge_detect` task, the description first requires outputs to return to zero after an edge, then states that they remain high until another edge. The reference RTL implements single-cycle pulses. In CVDP task `cvdp_copilot_sorter_0001`, the specification requires ascending order but does not define the element order on the packed output bus. The testbench assumes that the smallest element occupies the least-significant bits.

Adaptation and validation. We resolved these issues with explicit rules preserving each task's intent, then aligned the specification, Behavior reference and auxiliary RTL. We retained native interfaces and specified timing wherever possible, documenting substantive changes and their rationale. Adapted tasks underwent construction checks and human review in Appendices D.2 and D.4.

D.4 HUMAN REVIEW

Review scope. Researchers reviewed all 921 tasks across BEHAVE, VerilogEval-v2, RTLLM-v2 and CVDP with agent assistance, using the five criteria in Table 13. They checked task specifications and interfaces, inspecting source material, implementations and execution records where needed.

Revision and approval. Researchers made corrections directly or provided feedback to the agent for revision. Revised tasks repeated the applicable reconstruction and executable checks in Appendix D.2. A researcher approved the final version before B_1 was frozen as $B^\star$. Review records link each finding to its correction, checked version and final disposition.

Review outcomes. Table 13 classifies researcher-requested revisions by their primary reason, counting each revised task once. Table 14 summarizes recorded revision rounds across construction stages. Most Behavior comparisons required no revision.

Table 13: **Human review and revisions.** Each revised task is counted once by primary category.

Review item	Researcher checks	Tasks revised, n
Specification consistency	Requirements and examples agree, with valid inputs, parameter constraints and boundary behavior defined.	6
Numerical semantics	Widths, signedness, overflow, rounding, special values and operation order are explicit where relevant.	2
Interface and state	Packing, transaction counts and ordering, reset, backpressure and timing are defined where applicable.	14
Implementation fidelity	Behavior models and auxiliary RTL match the specified function, state transitions and memory effects.	7
Evidence integrity	Source and execution records match reviewed versions and runtime settings, including retests.	1
Total		**30**

Table 14: **Task revisions by construction stage.** Counts cover all 921 tasks: 600 BEHAVE tasks and 321 external-benchmark adaptations. Revised gives the number and percentage of revised tasks. The remaining columns group tasks by recorded feedback-driven revision rounds.

Stage	Revised, n (%)	0 rounds	1 round	≥ 2 rounds
B_2 comparison	6 (0.65%)	915	6	0
Agent review	14 (1.52%)	907	14	0
Human review	30 (3.26%)	891	29	1

D.5 DATA SPLITS AND RELEASE

Data splits. We reserve 10% of the tasks in each domain for evaluation, yielding 540 training tasks and 60 evaluation tasks with the same domain proportions. Both sets cover a range of difficulties. Tasks sharing a source implementation or differing only in parameters remain in the same split, together with their source materials. External benchmark adaptations are evaluation-only. Evaluation tasks and references are frozen before training and excluded from task acquisition.

Release. The release includes task specifications, golden Behavior models, auxiliary RTL and the interface information needed to run them. Source lists and review records document where each task came from, what was changed and which checks were run.

Licensing. Our original contributions will be released under Apache-2.0. Third-party materials retain their applicable licenses and required notices, documented in the task source list.

E Extended Related Work

LLMs for EDA. LLMs have been applied to a wide range of digital design and verification tasks, including RTL and testbench generation, debugging, and design optimization (Chang et al., 2023; Thakur et al., 2023; Yu et al., 2025; Qiu et al., 2025; 2024; 2026; Zhang et al., 2025b; Xin et al., 2026; Yan et al., 2025b; Mali et al., 2024; Orenes-Vera et al., 2023; Bhandari et al., 2024; Ma et al., 2024; Tsai et al., 2024; Yao et al., 2025; Yubeaton et al., 2026; Yu et al., 2026a; Islam et al., 2026; Chang et al., 2026; Wu et al., 2026a; Yan et al., 2025a; Zhang et al., 2026f; Ye et al., 2025b; 2026; Zhao et al., 2026b; Wang et al., 2025e; Tasnia et al., 2025; Wang et al., 2025f; Fang et al., 2026; Ping et al., 2026; Wang et al., 2026a; Thorat et al., 2025; Cuyckens et al., 2026; Chen et al., 2026a; Hu et al., 2025; Tan et al., 2026a; Cui et al., 2026; Hemadri et al., 2025). A parallel line develops training datasets and post-trains open-source LLMs for these tasks (Liu et al., 2024a;b; Thakur et al., 2024; Liu et al., 2025; Wang et al., 2025b; Yubeaton et al., 2025; Zhu et al., 2025a; Zhang et al., 2026b; Cui et al., 2024; Zhao et al., 2026a; Akyash et al., 2025; Calzada et al., 2025; Chen et al., 2026c; Wang et al., 2026b; Wei et al., 2025a; Zhang et al., 2025a; Deng et al., 2025; Zhang et al., 2026e; Deng et al., 2026; Chen et al., 2026b; Shi et al., 2025; Wang et al., 2025a; Fu et al., 2026; Wu et al., 2026b). Some methods also use high-level representations to support RTL training. BetterV translates RTL into cycle-accurate C models for paired instruction tuning (Pei et al., 2024). AutoVCoder uses Python execution to filter synthetic combinational RTL examples before supervised fine-tuning (Gao et al., 2024). BEHAVE instead uses executable behavior models to provide correctness rewards during RL.

Benchmarks for RTL Design and Verification. RTLLM (Lu et al., 2024; Liu et al., 2024c) and VerilogEval (Liu et al., 2023; Pinckney et al., 2025a) evaluate small module generation from natural-language specs, and CVDP (Pinckney et al., 2025b) extends the format to other RTL tasks such as debugging and comprehension. ArchXBench, NotSoTiny, and RealBench target larger or more realistic RTL designs (Purini et al., 2025; Ghorab et al., 2025; Jin et al., 2025). RTL-Repo evaluates repository-level code completion (Allam & Shalan, 2024), while CktEvo focuses on repository-level design evolution (Shi et al., 2026). ChipBench covers RTL generation, debugging, and reference-model generation (Yu et al., 2026c), while RTL-OPT targets RTL optimization (Lu et al., 2026b). IC-RTL and Pluto support PPA evaluation across different latencies, but rely on a dedicated testbench for each task (Hsin et al., 2026; Abdelatty et al., 2025). For evaluation, BEHAVE-Sim instantiates standardized testbenches from our dataset's golden behavior models and interface declarations, without requiring separately authored testbench code for each task.