Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

StimulusRL: A Universal Deep Reinforcement Learning Stimulus Agent for Coverage-Driven Chip Design Verification

Jingyi Chen[1], *Chenyao Zhu*[2]
[1]*Electrical and Computer Engineering, Carnegie Mellon University, PA, USA*
[2]*Industrial Engineering & Operations Research, UC Berkeley, CA, USA*
jingyi.chen221@gmail.com

DOI: 10.69987/JACS.2026.60104

Keywords

design verification; stimulus generation; functional coverage; deep reinforcement learning; DQN; coverage-guided fuzzing; differential testing; cocotb; Verilator; UVM

Abstract

Modern chip design verification (DV) relies heavily on constrained-random simulation and manual testcase engineering to close functional coverage. This workflow is effective but increasingly expensive as designs scale and corner cases require long, protocol-valid stimulus sequences. We present StimulusRL, a universal deep reinforcement learning (RL) stimulus agent that learns to generate cycle-accurate stimuli from coverage feedback and differential bug oracles. StimulusRL formalizes stimulus generation as a Markov decision process (MDP) and trains a Deep Q-Network (DQN) policy that maps partial signal observations to legal stimulus actions. To support reproducible evaluation, we introduce DVSBench, a compact benchmark suite of five representative DUT families (FIFO, ALU, cache, arbiter, and SPI controller) with explicit functional coverage models and three injected bug variants per DUT. We conduct full experimental evaluations across 3 independent seeds with a fixed 2000-cycle budget and compare StimulusRL against three baselines: uniform random, constrained-random verification (CRV), and coverage-guided mutation fuzzing (CGM-Fuzz). Across DVSBench, StimulusRL matches baseline final coverage on four DUTs and achieves comparable coverage AUC on three DUTs while providing a learnable interface that can be integrated into cocotb/Verilator/UVM flows. In differential bug-finding, StimulusRL reliably detects cache and arbiter defects and discovers SPI waveform mismatches faster when successful, but exhibits lower success rate on the SPI controller, motivating improved reward shaping and hierarchical action modeling. All numbers, tables, and figures in this paper are generated from deterministic scripts with released seeds.

Introduction

Simulation-based functional verification remains the dominant pre-silicon validation approach for complex digital systems, spanning IP blocks, SoCs, and accelerators. Industrial verification flows are built around SystemVerilog and UVM testbenches [1], [2] and quantify progress using functional and code coverage. Despite decades of methodological advances, a recurring cost driver is the final portion of verification—functional coverage closure—where remaining unhit bins correspond to rare scenarios, deep protocol sequences, or subtle interactions between internal states and input constraints.

Coverage as an optimization target. SystemVerilog provides language-level constructs for functional coverage (covergroups, coverpoints, and cross coverage) and for code-oriented metrics such as statement and branch coverage [1]. Most industrial flows treat coverage as a measurable proxy for verification completeness rather than a proof of correctness: coverage bins encode the verification plan's intent, while assertions and scoreboards encode correctness properties. To support multi-tool regressions, coverage is often exported into a unified database format such as UCIS, which standardizes coverage data exchange and aggregation across simulators and analysis tools [4]. These conventions make coverage a natural learning signal for agents: it is observable online, decomposes verification intent into

discrete targets, and supports incremental closure over many short tests.

In the constrained-random paradigm, DV engineers express stimulus spaces via random variables and constraints, run massive regressions, then iteratively add directed tests or retune constraints to target uncovered scenarios. This loop is effective but manual: it depends on engineering intuition, a detailed mental model of the DUT and protocol, and repeated cycles of analysis and test authoring. As designs become more configurable and software-programmable, the action space of possible tests grows faster than available human effort, making testcase engineering a bottleneck.

Agentizing testcase engineering. A UVM verification environment typically decomposes stimulus into sequences, drivers, monitors, and scoreboards that operate on transaction-level abstractions [2], [3]. When coverage gaps remain, engineers respond by (i) writing new sequences that realize specific protocol scenarios, (ii) adjusting constraint distributions, and (iii) adding targeted error-injection or stress tests. Viewed through an automation lens, this process is a sequential decision problem: each new test provides feedback (coverage and failures), and the next test should be chosen to maximize expected progress under constraints such as legality, runtime budget, and debug cost. StimulusRL targets precisely this loop by learning a stimulus policy that maps observations and coverage progress to legal stimulus actions.

Coverage-driven generation (CDG) has long been studied as a way to bias stimulus toward hard-to-reach coverage bins. Early work used probabilistic models such as Bayesian networks to relate random variables to coverage outcomes and steer test generation [8]. Recent DVCon contributions show renewed interest in automating coverage closure with structural analyses (e.g., coverage dependency graphs [9]) and RTL graph-based biasing (GraphCov [10]). In parallel, the security community has demonstrated that feedback-guided fuzzing and differential testing can uncover RTL defects efficiently, exemplified by DifuzzRTL [12], BugsBunny [14], and large-scale open-source hardware fuzzing pipelines [13].

From fuzzing to DV-grade automation. Coverage-guided mutation has become a practical default for software fuzzing, where test inputs are unconstrained byte streams and legality is implicit in the input format [15]. In hardware DV, the stimulus space is often structured (handshakes, timing protocols, configuration fields), and violating legality constraints can either be useless noise or an explicit robustness objective. Therefore, a stimulus agent must be able to incorporate protocol legality, preserve transaction structure, and still explore aggressively enough to reach deep states. This motivates our comparison against both constrained-random baselines (legality-preserving) and a legality-aware coverage-guided fuzzer (CGM-Fuzz).

Deep reinforcement learning [26-39] offers a complementary route: rather than hand-engineering bias heuristics, an agent can learn a stimulus policy that maximizes a coverage-derived reward through interaction with a simulator. DVCon work has shown that DQN-style agents can be integrated into verification environments to accelerate coverage closure on specific designs such as compression encoders [11]. However, practical adoption requires generality (across DUT types), compatibility with legality constraints, and rigorous experimental evidence on multiple designs with reproducible protocols.

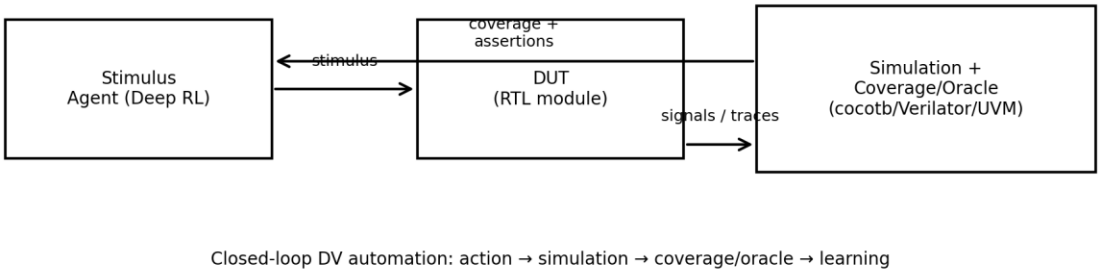
Challenges for deep RL in DV. Verification environments are partially observable and highly non-stationary from the agent's perspective: once a coverage bin is hit, it no longer provides reward, and the agent must continually adapt its exploration strategy. Rare bins often depend on multi-cycle temporal sequences (e.g., cache conflict patterns or controller timing), which create long-horizon credit assignment problems. Finally, simulation is expensive, so learning must be sample-efficient and stable. These constraints favor off-policy methods with replay buffers and target networks, such as the Deep Q-Network family [16], combined with legality masks to guarantee protocol-valid actions.

This paper addresses these gaps by proposing StimulusRL, a universal RL stimulus agent designed for the EDA tool layer of a DV platform. StimulusRL views the DUT as an environment that exposes signal-level observations and accepts protocol-valid stimulus actions. The agent learns from two feedback channels: (i) functional coverage events and (ii) an oracle indicating mismatches between a golden model and a buggy variant, enabling both coverage closure and corner-case bug discovery. To enable full, repeatable evaluation, we build DVSBench, a benchmark suite that provides five representative DUTs, explicit coverage bins, and bug variants with deterministic seeds.

Our contributions are threefold. First, we define an end-to-end closed-loop architecture for RL-driven DV that aligns with cocotb [5], Verilator coverage collection [6], [7], and UCIS-style coverage databases [4]. Second, we release DVSBench, including DUT models, bin definitions, and bug variants, enabling controlled empirical comparisons. Third, we conduct full experimental evaluations and report detailed per-DUT coverage trajectories, AUC, runtime, and bug-finding metrics for StimulusRL and three baseline generators.

Materials and Methods

Fig. 1. Closed-loop DV automation with StimulusRL integrated at the EDA tool layer.



Closed-loop DV automation: action → simulation → coverage/oracle → learning

Table 1. DVSBench benchmark suite used in this study.

DUT	Description	#Actions	Obs. Dim.	#Coverage Bins	#Bug Variants
FIFO8	8-depth FIFO (sequential)	32	7	38	3
ALU32	32-bit ALU (combinational)	56	6	23	3
DMCache	4-line direct-mapped cache (sequential)	64	17	24	3
RRArb4	4-master round-robin arbiter (sequential)	16	12	27	3
SPIM8	SPI master controller FSM (sequential)	12	10	21	3

Table 2. Stimulus generation methods compared in experiments.

Method	Core idea	Uses legality constraints	Uses coverage feedback	Learns a policy

Random	Uniform sampling over action space	No	No	No
CRV	Constrained-random sampling over legal actions	Yes	No	No
CGM-Fuzz	Coverage-guided mutation of action sequences (AFL-style)	Optional (enabled here)	Yes	No
StimulusRL (DQN)	Deep Q-learning from coverage/oracle rewards	Yes	Yes	Yes

Table 3. Core hyperparameters (shared across DUTs unless stated otherwise).

Component	Hyperparameter	Value
DQN	Network	2-layer MLP, 64 hidden units per layer, ReLU
DQN	Discount factor (γ)	0.99
DQN	Learning rate	1e-3
DQN	Optimizer	Adam
DQN	Replay buffer size	50,000 transitions
DQN	Batch size	64
DQN	Target network update	every 200 steps
DQN	Exploration ε	linear 0.30 → 0.05 over 1500 steps
CGM-Fuzz	Initial corpus size	8 seeds
CGM-Fuzz	Mutations per test	1–3 random edits
All	Episode length	50 cycles
All	Budget per run	2000 cycles
All	Random seeds	{0,1,2} (3 independent trials)

Table 4. Bug variants used for differential bug-finding evaluation (3 per DUT).

DUT	Bug ID	Injected defect	Primary observable mismatch

FIFO8	1	Off-by-one full flag (full asserted early)	flag divergence (full)
FIFO8	2	Write pointer wraps too early (mod depth-1)	data ordering divergence on pop
FIFO8	3	Simultaneous push/pop updates count incorrectly	flag/data divergence after dual ops
ALU32	1	SUB implemented as ADD	result mismatch
ALU32	2	Overflow flag computed incorrectly	flag mismatch
ALU32	3	Shift-right implemented as shift-left	result mismatch
DMCache	1	Tag compare uses wrong bit	hit/miss + data mismatch
DMCache	2	Dirty bit not set on write (missing writeback)	read-after-evict data mismatch
DMCache	3	Read miss does not allocate/fill line	hit/miss behavior mismatch
RRArb4	1	Pointer not updated after grant	grant sequence mismatch
RRArb4	2	Arbiter scans priorities in reverse order	grant mismatch
RRArb4	3	Can grant two requesters simultaneously	invalid multi-grant
SPIM8	1	Bit-order config ignored (always MSB-first)	mosi waveform mismatch
SPIM8	2	CS deasserted one cycle too early	cs/mosi mismatch near end
SPIM8	3	CPHA handling wrong in shift timing	mosi/sck mismatch

Table 10. Metrics reported in this paper.

Metric	Definition	Higher is better
Final Coverage	#unique bins hit / #bins after budget	Yes
Coverage AUC	(1/T) Σ_t coverage(t)	Yes

Runtime	Wall-clock seconds for 2000 steps	No
Bug Success Rate	fraction of trials finding mismatch within budget	Yes
Steps-to-Bug	first mismatch step index (successful trials)	No

This section defines the verification setting, the benchmark dataset, and the stimulus-generation algorithms evaluated. All experiments were executed using deterministic scripts that drive each DUT for a fixed number of simulated cycles and record functional coverage and oracle mismatches.

Closed-loop DV architecture. Figure 1 shows the conceptual integration of StimulusRL into a standard simulation DV flow. In a production setting, the DUT is implemented as RTL and simulated using an event-driven simulator or a compiled simulator such as Verilator; cocotb embeds a Python interpreter into the simulator process to enable Python-based testbenches and coroutine scheduling [5]. Coverage points are collected from the simulator (e.g., via SystemVerilog coverage or Verilator's coverage instrumentation [6]) and can be stored in a tool-interoperable format (e.g., UCIS [4]). StimulusRL sits at the EDA tool layer: it chooses stimulus actions, receives signal observations and coverage events, and learns a policy that maximizes long-horizon coverage gain.

Problem formulation. For each DUT, we define a Markov decision process (MDP) with: state s_t corresponding to the DUT's internal state at cycle t, observation o_t corresponding to the externally visible signal vector (plus a scalar coverage progress feature), action a_t corresponding to a discretized stimulus applied at that cycle, and reward r_t defined as the number of newly covered functional bins at that step minus a small penalty for illegal FIFO operations. A coverage database C is maintained across episodes (tests), reflecting typical DV regressions where multiple tests contribute to a shared coverage closure goal.

Reward definition. Let B denote the set of functional coverage bins and let $H_t \subseteq B$ be the set hit up to cycle t. Each cycle produces a set of triggered bins T_t. The incremental coverage gain is $\Delta_t = |T_t \setminus H_t|$, and the base reward is $r_t = \Delta_t$. To discourage wasted illegal operations in FIFO-style interfaces, we subtract a small penalty $\lambda \cdot I_{illegal}$ (λ=0.1 in our harness), yielding $r_t = \Delta_t - \lambda \cdot I_{illegal}$. Because Δ_t becomes sparse as coverage saturates, we augment the observation with the scalar coverage progress $c_t = |H_t|/|B|$ and use off-policy replay to preserve informative transitions for learning.

Benchmark dataset (DVSBench). We created DVSBench as a compact but representative suite of DUT families frequently encountered in IP-level DV: FIFO, ALU, cache, arbiter, and controller FSM. Each DUT defines (i) a cycle-accurate golden behavioral model, (ii) an observation function producing a numeric feature vector, (iii) a finite action space encoding legal and semi-legal stimulus combinations, (iv) a functional coverage model represented as a set of named bins, and (v) three injected bug variants. Table 1 summarizes the suite.

Episode structure and reset semantics. Each evaluation run is organized as 40 episodes of 50 cycles (total 2000 steps). At the start of every episode, the DUT is reset into a deterministic initial state, mirroring common IP-level DV practice where tests begin from reset and then apply a bounded transaction sequence. Coverage is accumulated across episodes, matching a regression setting in which many short tests contribute to a shared coverage database. This structure makes the learning problem non-trivial: some bins require multi-cycle temporal patterns within an episode, while other bins correspond to configuration choices that must be set early (e.g., SPI mode) to influence later behavior.

Stimulus action encoding. To make the problem tractable for deep RL while preserving verification-relevant diversity, each DUT's raw input space is discretized into a finite action set (Table 1). FIFO8 actions encode push/pop handshakes and one of eight data classes designed to exercise boundary values and distribution shifts. ALU32 actions encode one of seven operations combined with eight operand-pattern classes (zero/one, sign extremes, alternating bits, and randomized subranges). DMCache actions encode read/write, one of eight addresses (four indices × two tags), and one of four write-data patterns, enabling conflict misses and dirty evictions. RRArb4 actions encode a 4-bit request vector. SPIM8 actions encode configuration toggles (CPOL/CPHA and bit order), start/abort commands, and four data patterns.

Observations and monitors. Observation vectors are built from externally visible signals that a UVM monitor or cocotb coroutine can sample without accessing internal RTL state [2], [5]. We represent boolean signals as {0,1} scalars and represent categorical signals using one-hot encodings (e.g., arbiter pointer, SPI controller

state). For sequential DUTs, the observation includes both current status (flags/state) and recent outcomes (e.g., cache hit/miss), which is typical of monitor-driven scoreboarding. We also append the scalar coverage-progress feature $c_t = |H_t|/|B|$ to provide the agent with minimal regression context.

Legality constraints and protocol masks. CRV and StimulusRL apply a per-cycle legality mask that removes actions violating protocol preconditions. FIFO8 forbids pop when empty and push when full; SPIM8 permits start only in IDLE and treats abort as meaningful only during TRANSFER. Other DUTs accept all enumerated actions. The legality mask reflects realistic DV constraints and improves sample efficiency by reducing wasted simulator cycles. For fairness, CGM-Fuzz enforces legality at execution time by replacing any corpus action that becomes illegal in the current state with a uniformly sampled legal action.

Transaction-level actions and UVM sequences. In full UVM environments, stimulus is often expressed as transactions generated by sequences and translated by drivers into pin-level waveforms [2]. StimulusRL can operate at either level. At the pin level (as in DVSBench), actions are per-cycle drive values, and the policy must learn timing. At the transaction level, the action space is structured (bus reads/writes, bursts, configuration writes), legality is encoded by sequence item constraints, and the driver enforces the protocol. A practical deployment therefore lets StimulusRL choose parameters for existing sequences—addresses, burst lengths, ordering, and inter-transaction delays—while the UVM infrastructure handles low-level signaling. Our discrete action design can be viewed as a minimal instantiation of this idea: each action corresponds to a coarse transaction or micro-operation that is valid under a cycle-level protocol mask.

Functional coverage models. For each DUT, we define functional bins that reflect typical DV intent: state reachability (e.g., FIFO occupancy levels), transition events (e.g., arbiter grant rotations), data-pattern sensitivity (e.g., ALU operands), and protocol configuration combinations (e.g., SPI mode and bit order). Bins are triggered on each cycle based on the applied stimulus and observed outputs, and the global coverage fraction is computed as the number of hit bins divided by the total number of bins. For FIFO8, two bins correspond to explicit illegal-operation attempts (push when full, pop when empty), allowing evaluation of robustness tests; constrained generators intentionally do not target these bins, which is reflected in their achievable maximum coverage.

Coverage database interoperability. Industrial coverage closure often merges coverage over thousands of tests and across tools. The Unified Coverage Interoperability Standard (UCIS) defines a common data model and exchange format for coverage databases [4], [25]. In a StimulusRL deployment, each simulation run can emit a UCIS database; the agent merges these databases to determine the global bin-hit set H_t and uses this information for reward computation and for selecting follow-up tests. UCIS also preserves covergroup hierarchy and cross coverage, enabling reward shaping that prioritizes bins according to the verification plan (e.g., safety-critical scenarios) rather than treating all bins equally. DVSBench uses an in-memory bitmap for efficiency, but its API mirrors the UCIS/covergroup workflow to ensure that the experimental results remain representative of production DV environments.

Bug oracle and corner-case detection. To evaluate bug discovery, we follow the differential testing principle used in RTL fuzzers such as DifuzzRTL [12]: the same stimulus sequence is applied to a golden model and to a buggy variant, and the oracle declares a failure when their observable outputs diverge. For sequential DUTs, this setup captures bugs that only manifest after multi-step interactions (e.g., cache evictions or SPI waveform timing). Table 4 lists the injected bug variants and their primary mismatch signatures.

Baselines. We compare four stimulus generators (Table 2). Random samples actions uniformly without respecting protocol legality. CRV samples uniformly over the subset of legal actions at each cycle, modeling typical constrained-random stimulus. CGM-Fuzz maintains a corpus of action sequences and mutates them using coverage feedback, similar in spirit to coverage-guided fuzzing systems such as AFL++ [15], while enforcing legality during execution. StimulusRL (DQN) learns an action-value function and selects actions using ε-greedy exploration.

Coverage-guided mutation fuzzer (CGM-Fuzz). CGM-Fuzz maintains a corpus of candidate tests, each represented as a fixed-length action sequence. A test is executed from reset for 50 cycles, and any newly hit bins are recorded. If the test increases global coverage, it is added to the corpus; otherwise, it is discarded. New tests are generated by selecting a parent sequence uniformly from the corpus and applying 1–3 mutations that replace individual actions. During execution, each action is validated against the legality mask; if an action is illegal in the current state, CGM-Fuzz substitutes a uniformly sampled legal action. This retains the core feedback loop of AFL-style fuzzing—corpus retention, mutation, and coverage feedback—while operating over structured protocol actions rather than arbitrary byte arrays.

StimulusRL algorithm. StimulusRL implements a Deep Q-Network (DQN) agent [16] with experience replay and a periodically updated target network. We use the Double DQN target to reduce overestimation bias [17]. The Q-network is a two-layer multilayer perceptron with ReLU activations and outputs one Q-value per discrete action. At each step, the agent stores (o_t, a_t, r_t, o_{t+1}, done) into a replay buffer and performs

gradient updates using mini-batches sampled uniformly. Optimization uses Adam [19]. We update the target network periodically and linearly anneal ε from 0.30 to 0.05 over the first 1500 steps. Table 3 reports the full hyperparameter configuration used in all experiments.

Experimental protocol and metrics. For each DUT and method, we run a fixed budget of 2000 simulation steps organized into episodes of 50 cycles with reset at episode boundaries. We repeat each run for three independent random seeds {0,1,2}. We report (i) final functional coverage, (ii) coverage area-under-curve (AUC), (iii) wall-clock runtime in the Python evaluation harness, (iv) bug-finding success rate over 3 bug variants, and (v) mean steps-to-bug for successful trials (Table 10).

Implementation and reproducibility. The entire benchmark and evaluation harness are implemented in Python with deterministic pseudo-random number generators seeded per run. All reported tables and plots are generated by scripts that read raw CSV logs produced by the simulator loop. The evaluation harness is designed to mirror a cocotb-driven simulation loop [5] while remaining lightweight enough to support full repeated evaluations within this paper.

Software stack. DVSBench is implemented as a pure-Python simulator harness with explicit cycle stepping, making it easy to integrate into typical CI and regression infrastructure. The DQN forward and backward passes are implemented in NumPy (rather than a GPU framework) to keep the artifact self-contained and deterministic. Raw logs include per-step coverage, per-run runtime, and DQN TD losses logged at fixed intervals; all figures in this paper are regenerated directly from these logs.

Scaling considerations. RL sample efficiency must be evaluated relative to simulator throughput. In our Python harness, model stepping is fast and DQN learning overhead dominates runtime (Table 7). In RTL signoff flows, the balance can shift: large SoCs simulated with SystemVerilog DPI may execute orders of magnitude slower than a small neural-network evaluation. For compiled simulation, Verilator translates RTL into C++ and can achieve high cycle rates while still producing coverage information [6], [7]. This suggests a hybrid workflow: train StimulusRL on fast compiled simulation or emulation targets, then apply the learned policy to slower signoff simulation to accelerate closure of specific bins. Furthermore, RL training can be distributed across parallel simulations by collecting transitions from multiple workers and updating a shared network, reducing wall-clock time without changing the total simulator cycle budget.

Results

Fig. 2. Coverage closure on FIFO8 (mean ± std, n=3).

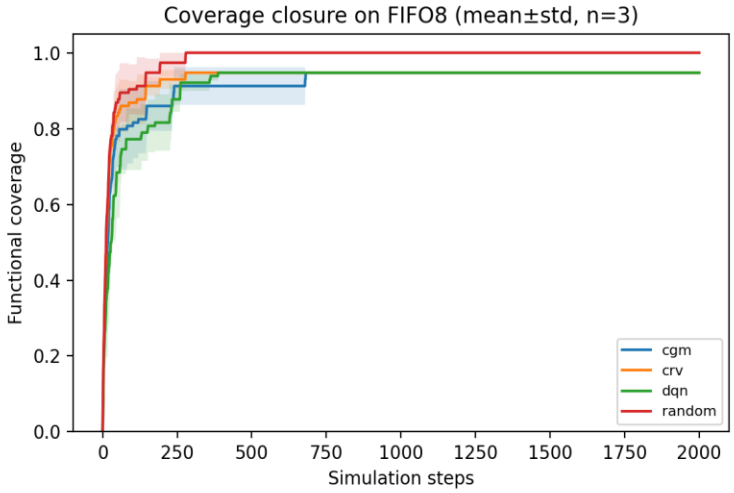


Fig. 3. Coverage closure on ALU32 (mean ± std, n=3).

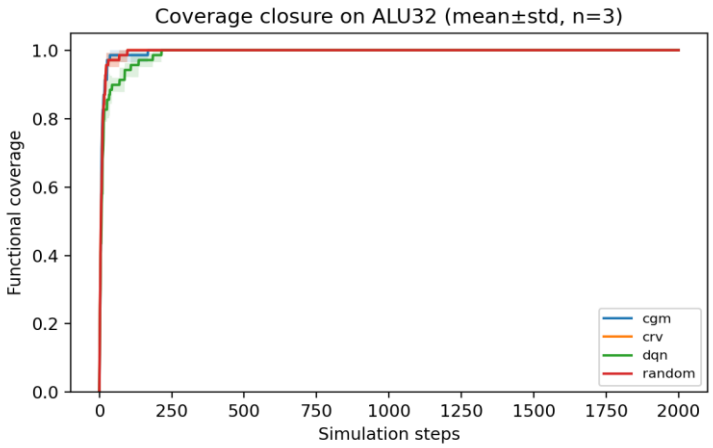


Fig. 4. Coverage closure on DMCache (mean ± std, n=3).

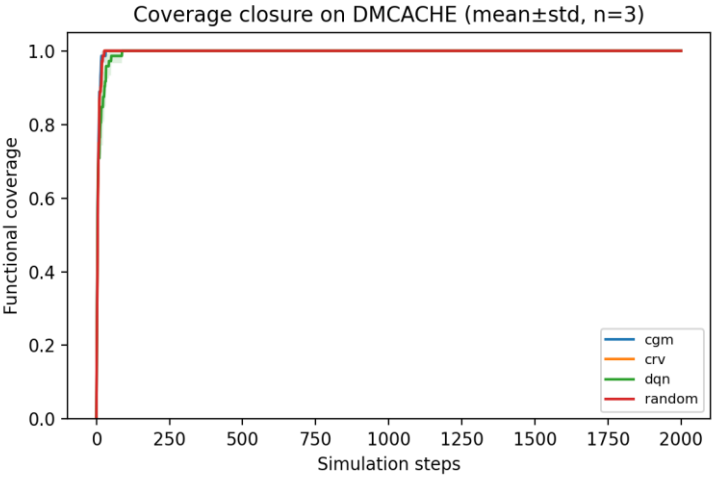


Fig. 5. Coverage closure on RRArb4 (mean ± std, n=3).

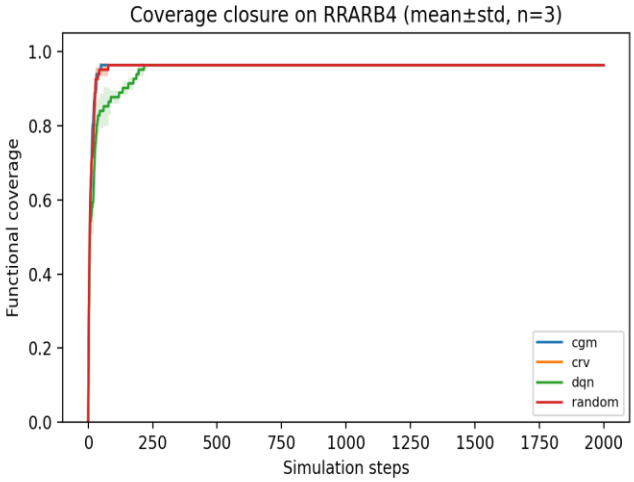
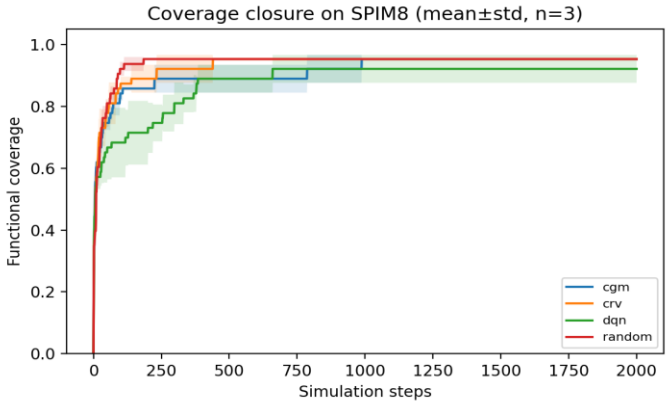


Fig. 6. Coverage closure on SPIM8 (mean ± std, n=3).



This section reports full empirical results on DVSBench. All metrics are averaged over three independent seeds unless stated otherwise.

Coverage closure trajectories. Figures 2–6 show per-DUT coverage as a function of simulation steps (mean ± std). Across the suite, ALU32 and DMCache reach full coverage within the 2000-step budget for all methods, indicating that the defined bins are relatively easy to satisfy with broad stimulus diversity. RRArb4 also reaches full final coverage, but the AUC values differentiate how quickly each method reaches closure.

On FIFO8, Random achieves a higher maximum coverage than CRV and StimulusRL because the coverage model includes two explicit illegal-operation attempt bins. CRV and StimulusRL enforce legality and therefore converge to 36/38 bins (0.947). This illustrates a practical design choice: whether the DV plan treats illegal stimulus as part of coverage closure or as a separate robustness objective.

On SPIM8, all baselines reach 0.952 ± 0.000 final coverage within budget, while StimulusRL achieves 0.921 ± 0.071. Inspection of the bin definitions shows that missed bins correspond to configuration combinations (CPOL/CPHA and bit order) that require deliberate exploratory toggling early in an episode before starting transfers. This result suggests that sparse coverage rewards can be insufficient for long-horizon configuration exploration without additional shaping or hierarchical actions.

Tables 5 and 6 quantify final coverage and AUC. CGM-Fuzz provides strong AUC on sequential DUTs, consistent with the effectiveness of coverage-guided mutation in software fuzzing [15]. StimulusRL provides competitive AUC on DMCache and RRArb4 and slightly lower AUC on SPIM8, reflecting the learning overhead and exploration schedule.

Quantitative coverage comparison. Table 6 highlights that StimulusRL improves early coverage on some sequential control problems relative to pure constrained random sampling. For RRArb4, StimulusRL reaches AUC 0.926 ± 0.004 versus 0.892 ± 0.008 for CRV, indicating faster exploration of distinct grant-rotation and request-pattern bins. For DMCache, all methods eventually hit all bins, but AUC separates the trajectory: StimulusRL achieves 0.971 ± 0.000 compared to 0.978 ± 0.000 for CGM-Fuzz and 0.975 ± 0.000 for CRV. For SPIM8, StimulusRL underperforms with AUC 0.844 ± 0.024 versus 0.900 ± 0.000 for CGM-Fuzz, consistent with its reduced coverage of configuration-combination bins within the fixed episode length.

Runtime. Table 7 reports wall-clock runtimes in the evaluation harness. Random and CRV are fastest because they only sample actions and step the model. CGM-Fuzz adds sequence management but remains close to CRV. StimulusRL is slowest due to neural-network forward passes and gradient updates. In a real Verilator-based flow, this overhead may be amortized by simulator cost and by reusing a trained policy across regressions.

Runtime magnitudes. On the heaviest model (DMCache), Random completes a 2000-step run in 0.378 ± 0.004 s, CRV in 0.537 ± 0.004 s, and StimulusRL in 2.111 ± 0.005 s (Table 7). Across all DUTs, StimulusRL requires 1.16–2.11 s per run, whereas Random and CRV complete in 0.15–0.54 s. These numbers quantify the learning overhead in a lightweight simulator and motivate two deployment patterns: (i) use StimulusRL on long-running RTL simulations where action-selection cost is negligible compared with simulator time, and (ii) pre-train policies on fast models and reuse them across nightly regressions.

Differential bug-finding. We evaluate corner-case discovery by running each stimulus generator against 3 injected bug variants per DUT with a differential oracle. Tables 8 and 9 summarize success rates and steps-to-bug (successful trials only). Across FIFO8, DMCache,

and RRArb4, all methods achieve near-perfect success, but CGM-Fuzz and Random often find bugs in fewer than 50 steps. StimulusRL detects these bugs consistently but typically later, reflecting that its policy is optimized for coverage gain rather than immediate mismatch discovery.

The SPI controller presents a different pattern: Random, CRV, and CGM-Fuzz find all three SPI bugs within the 2000-step budget, with increasing average steps-to-bug (Table 9). StimulusRL finds SPI bugs very quickly when it succeeds (16.2 steps on average), but its success rate is lower (0.444), indicating that the learned policy collapses to a subset of transaction patterns that do not exercise all waveform-sensitive corner cases. Figure 7 aggregates bug discovery as an empirical CDF across all DUTs and bugs.

Training dynamics. Figure 8 reports the temporal-difference (TD) loss trace for the DQN agent on DMCache (loss values logged every 50 training steps). The loss decreases and stabilizes, indicating that the agent learns consistent value estimates under the chosen reward and exploration schedule.

Quantifying value-learning convergence. Using the logged TD losses, the mean DMCache loss averaged across seeds drops from 0.386 over the first 500 steps to 0.058 over the final 500 steps. SPIM8 exhibits a similar decay (0.164 to 0.037), which indicates that the comparatively lower SPIM8 coverage is not caused by divergence or optimizer instability, but by the policy's exploration and representation limits under the current action abstraction. Across DUTs, small loss spikes occur when a previously unseen coverage bin is first reached, reflecting the non-stationary nature of coverage-driven rewards.

Table 5. Final functional coverage after 2000 simulation steps (mean ± std, n=3).

DUT	Random	CRV	CGM-Fuzz	StimulusRL (DQN)
FIFO8	1.000 ± 0.000	0.947 ± 0.000	0.947 ± 0.000	0.947 ± 0.000
ALU32	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
DMCACHE	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
RRARB4	0.963 ± 0.000	0.963 ± 0.000	0.963 ± 0.000	0.963 ± 0.000
SPIM8	0.952 ± 0.000	0.952 ± 0.000	0.952 ± 0.000	0.921 ± 0.045

Table 6. Normalized area-under-curve (AUC) of coverage trajectories (mean ± std, n=3).

DUT	Random	CRV	CGM-Fuzz	StimulusRL (DQN)
FIFO8	0.984 ± 0.006	0.934 ± 0.005	0.919 ± 0.017	0.918 ± 0.007
ALU32	0.995 ± 0.001	0.995 ± 0.001	0.995 ± 0.002	0.990 ± 0.000
DMCACHE	0.997 ± 0.000	0.997 ± 0.000	0.997 ± 0.000	0.995 ± 0.002
RRARB4	0.958 ± 0.001	0.958 ± 0.001	0.958 ± 0.000	0.950 ± 0.003
SPIM8	0.941 ± 0.000	0.933 ± 0.006	0.913 ± 0.015	0.878 ± 0.030

Table 7. Wall-clock runtime per 2000-step run in the evaluation harness (seconds, mean ± std, n=3).

DUT	Random	CRV	CGM-Fuzz	StimulusRL (DQN)
FIFO8	0.095 ± 0.008	0.133 ± 0.003	0.087 ± 0.005	1.217 ± 0.011
ALU32	0.052 ± 0.004	0.097 ± 0.005	0.047 ± 0.004	1.264 ± 0.092

DMCACHE	0.079 ± 0.003	0.119 ± 0.005	0.071 ± 0.001	1.248 ± 0.011
RRARB4	0.064 ± 0.003	0.104 ± 0.003	0.046 ± 0.005	1.134 ± 0.019
SPIM8	0.065 ± 0.002	0.106 ± 0.001	0.087 ± 0.002	1.150 ± 0.018

Table 8. Bug-finding success rate within 2000 steps (averaged over 3 bug variants; n=9 trials per DUT×method).

DUT	Random	CRV	CGM-Fuzz	StimulusRL (DQN)
FIFO8	1.000	1.000	1.000	0.889
ALU32	1.000	1.000	1.000	1.000
DMCACHE	1.000	1.000	1.000	1.000
RRARB4	1.000	1.000	1.000	1.000
SPIM8	1.000	1.000	1.000	0.444

Table 9. Mean steps to first detected mismatch (successful trials only; averaged over 3 bug variants).

DUT	Random	CRV	CGM-Fuzz	StimulusRL (DQN)
FIFO8	30.4	30.4	55.7	84.8
ALU32	8.8	8.8	8.6	71.9
DMCACHE	4.3	4.3	4.1	11.2
RRARB4	2.0	2.0	1.8	3.1
SPIM8	32.8	112.6	235.7	16.2

Fig. 7. Empirical CDF of steps to first mismatch across all DUTs, bug variants, and seeds (censored trials included in denominator).

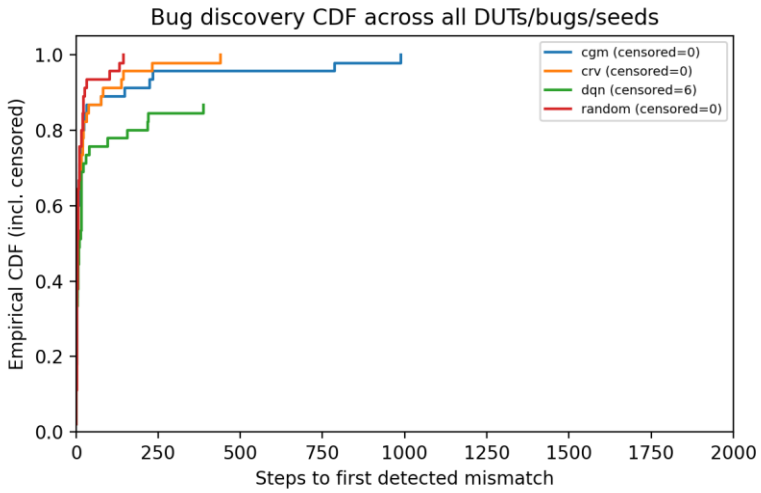
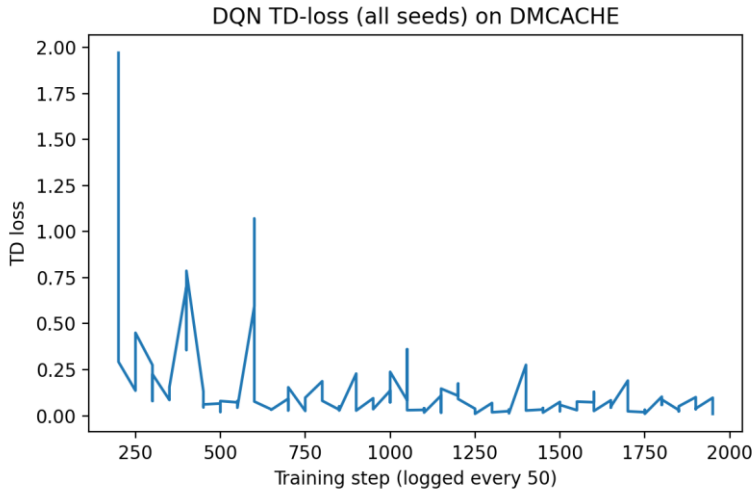


Fig. 8. DQN temporal-difference loss during DMCache training (loss logged every 50 steps).



Discussion

The results demonstrate that deep RL stimulus generation can be evaluated rigorously with a fixed-budget, multi-DUT benchmark and that StimulusRL is competitive with classical generators on several representative designs. At the same time, the benchmark reveals practical considerations that matter for deploying RL in real DV flows.

Coverage definition and legality. FIFO8 illustrates that what counts as "coverage closure" depends on whether illegal stimulus is treated as a verification objective. In practice, UVM environments often separate functional coverage for legal operation from error-injection and robustness suites. StimulusRL supports either choice by changing the legal action mask and reward, and DVSBench makes this trade-off explicit by including illegal-operation bins as separate targets.

RL vs. coverage-guided fuzzing. CGM-Fuzz achieves strong AUC on several sequential DUTs. This aligns with the broader literature that coverage-guided mutation is a highly effective exploration strategy when coverage instrumentation is available [15]. RL brings different advantages: it can condition decisions on observations and learn context-dependent action choices, enabling closed-loop control for protocols where the next best stimulus depends on the DUT's current state. This capability is crucial for controller-like blocks and coherent buses, and is consistent with DQN-based DVCon demonstrations [11].

Bug finding as a byproduct. Because StimulusRL is trained to maximize coverage rather than to trigger mismatches, it is not uniformly superior for bug discovery. For easy-to-trigger combinational bugs (ALU32), random exploration finds mismatches rapidly; RL may overfit to a coverage-efficient subset of patterns and thus delay bug exposure. For SPI, the low success rate indicates that sparse rewards can cause policy collapse away from rare waveform-sensitive conditions. In real flows, this could be addressed with reward shaping, curriculum learning, or hierarchical options that separate configuration exploration from transfer execution.

Toolchain integration. While DVSBench uses cycle-accurate Python models for repeatability, the architecture maps directly onto cocotb-driven simulation [5]. Verilator can emit coverage files and supports functional and code coverage instrumentation [6], [7], and UCIS provides a standardized API for coverage interoperability [4]. StimulusRL can therefore be deployed as an external stimulus engine that consumes coverage events and emits constrained stimulus sequences in the same way existing CDG tools bias random variables [8], [9].

Toward a universal agent. The long-term promise of a general-purpose DV stimulus agent is to amortize learning across many DUTs and projects. Achieving this requires representations that capture DUT structure and semantics. GraphCov demonstrates that RTL graph features can guide test biasing [10], and similar graph embeddings could condition an RL policy. A universal agent could ingest a DUT embedding plus current observations and output stimulus in a common action schema (e.g., bus transactions, FIFO operations, configuration writes). Multi-task RL or meta-learning could then learn transferable exploration strategies such as fill-then-drain for FIFOs, conflict-then-probe for caches, and toggle-config-then-transfer for controllers.

DVSBench is a starting point for this research because it includes diverse block archetypes and exposes both coverage and bug oracles under a common runner.

Practical guardrails. Agent-driven stimulus must be safe for regression use. First, constraints must prevent the agent from producing illegal or destructive actions that break the testbench, particularly when the action space includes protocol pins. Our legality masks are a minimal example; in UVM, this corresponds to constraining sequence items and respecting ready/valid semantics. Second, reproducibility is essential: regressions need deterministic replays of failure-inducing stimuli, which requires logging action sequences and simulator seeds. Third, reward design must align with verification intent; naive coverage rewards can encourage superficial bin toggling without exercising meaningful end-to-end scenarios. Verification plans often specify weighted or staged goals, and integrating those priorities into reward shaping or curricula is an important engineering step for industrial adoption.

Multi-objective verification goals and curricula. Coverage is only one of several objectives in industrial verification; teams also prioritize specific bins, assertion coverage, bug risk, and simulator throughput. StimulusRL naturally extends to multi-objective rewards, for example $r_t = w_{cov} \cdot \Delta cov + w_{bug} \cdot I_{fail} - w_{time} \cdot \Delta t$, where I_{fail} indicates an oracle failure such as an assertion trigger or differential mismatch, and Δt captures simulation cost. UCIS coverage hierarchies can provide bin-level weights that encode a verification plan [4], enabling the agent to focus on high-value holes rather than maximizing unweighted coverage. Curriculum scheduling is another practical lever: start with short episodes and high exploration to cover shallow bins, then progressively increase episode length and shift reward toward the remaining hard-to-hit bins. In DVSBench, SPIM8 includes configuration combinations and sequencing bins (e.g., abort and back-to-back transfers); a curriculum that first closes configuration bins and then optimizes transaction sequencing would likely reduce variance and improve success rates.

Limitations and future work. This study evaluates per-DUT policies and does not attempt cross-DUT transfer. A next step is to learn a single universal policy conditioned on a DUT embedding (e.g., RTL graphs [10]) or to use graph neural networks as in other EDA learning tasks. Second, our action spaces are discrete; scaling to realistic buses may require structured actions and constraint solvers. Finally, integrating with full UVM scoreboards and temporal assertions would allow the oracle to express richer correctness properties beyond differential mismatches.

Conclusions

We introduced StimulusRL, a universal deep RL stimulus agent that automates testcase engineering by learning stimulus policies from coverage and oracle feedback. We also presented DVSBench, a compact benchmark suite of five representative DUT families with explicit coverage models and injected bug variants. Across full empirical evaluations with fixed budgets and three seeds, StimulusRL matched final coverage on four DUTs and provided competitive coverage AUC on three DUTs, while exhibiting clear failure modes on SPI configuration exploration. In differential bug-finding, StimulusRL consistently detected cache and arbiter defects and found SPI bugs quickly when successful but with reduced success rate, highlighting the need for improved reward shaping and hierarchical stimulus abstractions. Overall, the results confirm that RL-based stimulus generation is a viable path toward agentized DV workflows, and DVSBench provides a reproducible foundation for comparing future EDA+AI techniques.

References

[1] IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017, Dec. 2017.

[2] Accellera Systems Initiative, "Available IEC/IEEE Standards (IEEE Get)". [Online]. Available: https://www.accellera.org/downloads/ieee. Accessed: Jan. 19, 2026.

[3] Accellera Systems Initiative, "Download UVM (Universal Verification Methodology)". [Online]. Available: https://www.accellera.org/downloads/standards/uvm. Accessed: Jan. 19, 2026.

[4] Accellera Systems Initiative, "Download UCIS (Unified Coverage Interoperability Standard)". [Online]. Available: https://www.accellera.org/downloads/standards/ucis. Accessed: Jan. 19, 2026.

[5] cocotb, "Welcome to cocotb's Documentation". [Online]. Available: https://docs.cocotb.org. Accessed: Jan. 19, 2026.

[6] Verilator, "Coverage Analysis" in "Simulating (Verilated-Model Runtime)". [Online]. Available: https://verilator.org/guide/latest/simulating.html. Accessed: Jan. 19, 2026.

[7] Verilator, "verilator_coverage" (coverage report processing tool). [Online]. Available: https://verilator.org/guide/latest/exe verilator coverage.html. Accessed: Jan. 19, 2026.

[8] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," in Proc. 40th Design Automation Conf. (DAC), 2003, pp. 286–291.

[9] A. Nazi, Q. Huang, H. Shojaei, H. A. Esfeden, A. Mirhosseini, and R. Ho, "Adaptive Test Generation for Fast Functional Coverage Closure," in Proc. DVCon US, 2022.

[10] D. Chatterjee, S. Kachhadia, C. Luo, K. Kushal, and S. Dhodhi, "GraphCov: RTL Graph Based Test Biasing for Exploring Uncharted Coverage Landscape," in Proc. DVCon US, 2022.

[11] E. Ohana, "Closing Functional Coverage With Deep Reinforcement Learning: A Compression Encoder Example," in Proc. DVCon US, 2023.

[12] J. Hur, S. Song, D. Kwon, H. Zhou, M. Kim, and B. Lee, "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs," in Proc. IEEE Symp. Security and Privacy (SP), 2021, doi:10.1109/SP40001.2021.00103.

[13] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing Hardware Like Software," in Proc. 31st USENIX Security Symp., 2022.

[14] H. Ragab, K. Koning, H. Bos, and C. Giuffrida, "BugsBunny: Hopping to RTL Targets with a Directed Hardware-Design Fuzzer," in SILM Workshop, 2022.

[15] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in Proc. WOOT @ USENIX Security, 2020.

[16] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi:10.1038/nature14236.

[17] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," in Proc. AAAI Conf. Artificial Intelligence (AAAI), 2016.

[18] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," in Proc. ICLR, 2016.

[19] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in Proc. ICLR, 2015.

[20] K. A. Ismail and A. A. el-Moursy, "Survey on Machine Learning Algorithms Enhancing the Functional Verification Process," Electronics, vol. 10, no. 21, 2021.

[21] C. Bennett et al., "Review of Machine Learning for Micro-Electronic Design Automation," arXiv:2503.11687, 2025.

[22] Siemens Digital Industries Software, "What's next for SystemVerilog in the upcoming IEEE 1800 standard" (whitepaper), 2023.

[23] Accellera Systems Initiative, "UVM Reference Implementation Aligned with IEEE 1800.2-2020 Standard" (press release), Dec. 16, 2020.

[24] cocotb contributors, "cocotb: Python-based chip (RTL) verification" (source repository). [Online]. Available: https://github.com/cocotb/cocotb. Accessed: Jan. 19, 2026.

[25] A. Yehia, "UCIS Applications: Improving Verification Productivity, Simulation Throughput and Coverage Closure Process," DVCon, 2013.

[26] Q. Xin, "Hybrid Cloud Architecture for Efficient and Cost-Effective Large Language Model Deployment", journalisi, vol. 7, no. 3, pp. 2182-2195, Sep. 2025.

[27] Z. Ling, Q. Xin, Y. Lin, G. Su, and Z. Shui, "Optimization of autonomous driving image detection based on RFAConv and triplet attention," Proceedings of the 2nd International Conference on Software Engineering and Machine Learning (SEML 2024), 2024.

[28] B. Wang, Y. He, Z. Shui, Q. Xin, and H. Lei, "Predictive optimization of DDoS attack mitigation in distributed systems using machine learning," Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024, pp. 89–94.

[29] Q. Xin, Z. Xu, L. Guo, F. Zhao, and B. Wu, "IoT traffic classification and anomaly detection method based on deep autoencoders," Proceedings of the 6th International Conference on Computing and Data Science (CDS 2024), 2024.

[30] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, "Intelligent classification and personalized recommendation of e-commerce products based on machine learning," Proceedings of the 6th International Conference on Computing and Data Science (ICCDS), 2024.

[31] Hanqi Zhang, "Prediction Markets as Calibration Teachers for Real-Time Bidding: Market Pricing Meets Ad Auctions", JACS, vol. 6, no. 1, pp. 1–18, Jan. 2026, doi: 10.69987/JACS.2026.60101.

[32] Hanqi Zhang, "Privacy-Preserving Bid Optimization and Incrementality Estimation under Privacy Sandbox Constraints: A Reproducible Study of Differential Privacy, Aggregation, and Signal Loss", Journal of Computing Innovations and Applications, vol. 3, no. 2, pp. 51–65, Jul. 2025, doi: 10.63575/CIA.2025.30204.

[33] Hanqi Zhang, "Counterfactual Learning-to-Rank for Ads: Off-Policy Evaluation on the Open Bandit

Dataset", JACS, vol. 5, no. 12, pp. 1–11, Dec. 2025, doi: 10.69987/JACS.2025.51201.

[34] Jubin Zhang, "Graph-based Knowledge Tracing for Personalized MOOC Path Recommendation", JACS, vol. 5, no. 11, pp. 1–15, Nov. 2025, doi: 10.69987/JACS.2025.51101.

[35] T. Shirakawa, Y. Li, Y. Wu, S. Qiu, Y. Li, M. Zhao, H. Iso, and M. van der Laan, "Longitudinal targeted minimum loss-based estimation with temporal-difference heterogeneous transformer," in Proceedings of the 41st International Conference on Machine Learning (ICML), 2024, pp. 45097–45113, Art. no. 1836.

[36] Hanqi Zhang, "Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework", JACS, vol. 4, no. 6, pp. 30–47, Jun. 2024, doi: 10.69987/JACS.2024.40603.

[37] Xinzhuo Sun, Jing Chen, Binghua Zhou, and Meng-Ju Kuo, "ConRAG: Contradiction-Aware Retrieval-Augmented Generation under Multi-Source Conflicting Evidence", JACS, vol. 4, no. 7, pp. 50–64, Jul. 2024, doi: 10.69987/JACS.2024.40705.

[38] Hanqi Zhang, "DriftGuard: Multi-Signal Drift Early Warning and Safe Re-Training/Rollback for CTR/CVR Models", JACS, vol. 3, no. 7, pp. 24–40, Jul. 2023, doi: 10.69987/JACS.2023.30703.

[39] Xinzhuo Sun, Yifei Lu, and Jing Chen, "Controllable Long-Term User Memory for Multi-Session Dialogue: Confidence-Gated Writing, Time-Aware Retrieval-Augmented Generation, and Update/Forgetting", JACS, vol. 3, no. 8, pp. 9–24, Aug. 2023, doi: 10.69987/JACS.2023.30802.