International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

CROSS-LAYER HW–SW CO-VERIFICATION USING C-TEST INTEGRATION IN UVM ENVIRONMENTS

Aparna Mohan[1]

[1]North Carolina State University, Raleigh, North Carolina

Abstract

Contemporary SoCs require the addition of comprehensive cross-layer verification to integrate hardware and embedded software test cases to discover bugs in the hardware-software integration that can not be identified by live tests in either domain. Traditional verification processes tend to break RTL functional verification and embedded software validation into independent processes causing blind spots to surface late in the silicon lifecycle. The following paper describes a simple way of including C-based software tests into UVM-based hardware testbenches to provide unified stimulus generation, better coverage closure and more realistic corner-case verification. We outline the infrastructure needed, the way promising virtual registers are designed, and how the means of synchronization with their help are achieved so as to enable coherent HW-Software co-verification. The measures in coverage and bug detection are shown by a real-world case study of an AXI-based SoC. They provide lessons learned and best practices in order to facilitate the adoption of this methodology at scale by the design teams.

Keywords: Co-verification, UVM, C-test, cross-layer, SoC, functional coverage, HW–SW integration.

1. Introduction

The modern System-on-Chip (SoC) architecture has slowed down the aspects of hardware and software interaction, which are increasingly more complex in scaling. Whether it is the high-performance CPUs managing DMA engines to security coprocessors loading cryptographic keys, the correctness of the system at this level requires a smooth interplay of domains, both hardware and firmware (Hennessy & Patterson, 2018).

Industry experience however demonstrates that the older hardware verification methods, involving either constrained-random UVM testbenches or protocol checkers, are not enough to find real integration bugs when the code running on the software can affect the hardware in ways that testbenches cannot adequately exercise (Foster, 2015). Similarly it is arguably possible to imagine that pure software validation on abstract virtual platforms will exercise low-level hardware corner cases, e.g., register resets, protocol deadlocks, or memory-mapped peripheral race conditions (Bergeron, 2006).

To fill this gap an increased interest in cross-layer HW-SW co-verification has emerged, where C-based firmware is executed in parallel with RTL simulation. According to this implementation, C-tests (or high-level software tests): may setup initial conditions of

systems, generate bus traffic and check the expected behavior along with stimuli and monitors based on UVM (Cadence, 2021).

The industry de facto standard modular and reusable block-level and SoC testbenches are Universal Verification Methodology (UVM). The adding of C-tests to UVM environments helps to eliminate the final mile between pre-silicon functional coverage and post-silicon verification (Goel & Foster, 2015). It enables:

- Real-life initialization sequences.
- Edge-cases controlled by firmware.
- Cross domain coverages collection.

The implementation of C-tests within a UVM simulation however is not trivial. It has to bootstrap embedded code in the RTL CPU model, respond to HW stimulus and SW driven triggers, and monitor end-to-end outcomes. Practical questions many design teams have to ask themselves is how do we interconnect UVM sequences with C-test drivers? What do we do to synchronize checkers? What do we do to reuse the same tests to emulate or bring-up silicon?

In this paper I will be answering these questions by:

1. Proposing a scalable co-verification model that will integrate the C-tests with UVM agents through the virtual interfaces between register interfaces.
2. Describing a reusable testbench framework in which HW-SW synchronization is carried out.
3. Proving the advantages through industrial SoC example.
4. Exchange of the best practices and limitations and future plans.

2. Background and Related Work

2.1 HW–SW Co-Verification: Why It Matters

The earlier designs of SoC were able to verify hardware and software in isolation due to the ease of integration points, there were mainly configuration registers or boot vectors (Hennessy & Patterson, 2018). Nowadays clean separation is unrealistic with complex bus fabrics, multi-core coherency, secure enclave, and privilege levels at run time.

The reports of academic studies and the reports of the industry constantly demonstrate that weird bugs, such as side effects on a register, handshake races, or insecurity leaks, are frequently covered by situations that can only be effortlessly provoked by the real software (Holler et al., 2020). It is commonly seen that such bugs remain out of simulation, and only come back as failure after manufacturing.

2.2 The UVM Standard

Universal Verification Methodology (UVM) has replaced Verilog as the framework of choice to create reusable, object-oriented and tested, testbenches (Bergeron, 2006). It

International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

standardizes:

- Bus protocols agents and drivers.
- Virtual sequencers to orchestrate a transaction.
- A scoreboard to check the correctness of data.
- Sign- off measure coverage collectors.

UVM is excellent at the generation of constrained-random traffic and functional coverage checking, but by itself presumes that the only source of stimulus is the hardware. As a result of a lack of real software on the core, there are numerous integration scenarios that cannot be ascertained (Goel & Foster, 2015).

2.3 What Are C-Tests?

C-tests are top-level test routines in embedded C compiled to the target CPU and running on the hardware simulation. These tests include;

- Peripheral registers configuration.
- Trigger interrupts.
- commence or abort data transfers.
- Verify the values used by registers or memory.

In the past, C-tests were tried on FPGA prototypes or silicon bring-up boards (Cadence, 2021). When they are bridged into a pre-silicon UVM testbench, these real-world flows enter into RTL simulation.

2.4 Related Work

Previous approaches to the HWSW co-verification were:

- Co-simulation by means of ISS (Instruction Set Simulators) connected to RTL (Siegmund et al., 2008).
- The approximate timing of Transaction-Level Modeling (TLM) (Ghenassia, 2005).
- Hybrid- Hybrid emulation flows using software on the FPGA (Cadence Emulation Whitepaper, 2020).

The approaches however come at the expense of cycle accuracy or increase complexities in modeling. An immediate combination of compiled C-tests within the identical RTL simulation eliminates this gap yet calls for a strong synchronization and observing to line-up the software activity with UVM drivers.

Received: August 02, 2025

International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

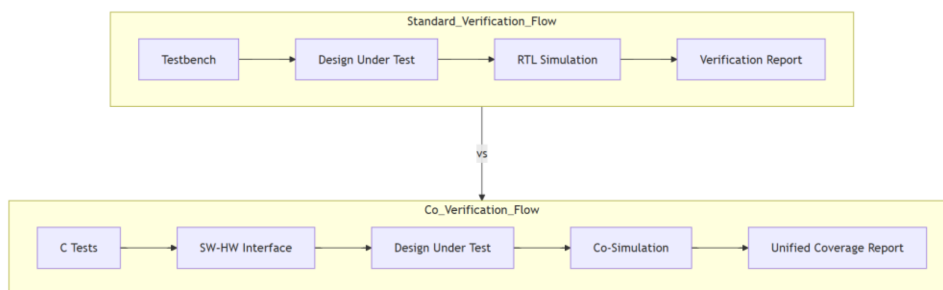


Figure 1. Standard vs. Co-Verification Flow

Table 1

3.

Verification Flow	Stimulus Source	Coverage	Integration Bugs Found
RTL-Only (UVM)	Randomized sequences	High for block-level	Misses HW–SW edge cases
SW-Only (Virtual Platform)	Boot firmware	High for SW logic	Misses hardware RTL bugs
HW–SW Co-Verification	Both C-test & UVM	High for cross-layer	Captures integration bugs

Problem Statement and Motivation

The same issue plagues the work of the modern SoC verification teams: when the functional sign-off has been performed, the last integration of the hardware and software may be debugged after silicon. Although UVM testbenches will be sophisticated, corner-case errors passing through needing actual execution of the firmware, delay the project and result in costly re-spins (Hennessy & Patterson, 2018; Foster, 2015).

Common forms of them are:

- Tiny register set-ups that could only be seen by genuine boot code.
- The bugs in the interrupt controllers were only an outcome of particular SW combinations.
- DMA or the bus locks late when SW engages in sequences that I did not think of in the testbench.
- Configuration of security bugs that may include a privilege escalation path which is not subject to block-level checks (Holler et al., 2020).

Findings Traditional constrained-random stimulus in UVM is effective at protocol and transaction validation, it is fundamentally disconnected with the logic being run by production firmware. Conversely, pure SW-only compared on virtual platforms do not support cycle-precise hardware characteristic hence timing-sensitive corner bugs go undetected (Siegmund et al., 2008; Ghenassia, 2005).

ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

Motivating Scenario:

Take a system on a chip having an ARM Cortex-M CPU, AXI bus, DMA engines, and a variety of memory-mapped peripherals.

- Bus fabric can be driven to check transactions with directed sequences by using the UVM testbench.

In the meantime, the real C firmware is initializing peripherals, configuring DMA, using interrupts and performing processing.

Integration bugs that require a real boot order, require races or require resetting of registers, may persist to first silicon in case such flows are tested separately.

This can be addressed by a cross-layer co-verification flow.

1. Running the simulation with the true compiled real C code in the CPU.
2. Using UVM to connect that SW implementation to the RTL testbench.
3. Ensuring that HW and SW flows are in sync and addressed.

In the absence of this, sign-off is not complete and the costly silicon bugs are still likely to occur (Cadence, 2021).

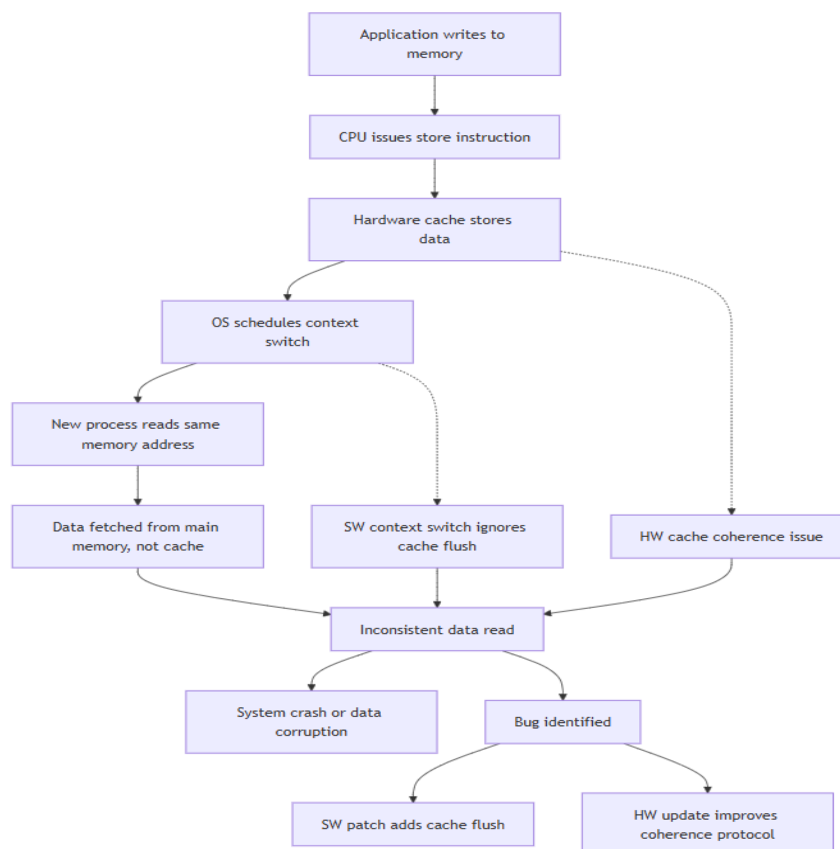


Figure 2. Real-World HW–SW Bug Example

A simple schematic shows an ARM CPU booting, configuring a DMA, triggering an AXI

Received: August 02, 2025

International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

transfer that deadlocks the bus due to a missing handshake. The testbench alone does not find this bug — a C-test does.

4. Proposed Co-Verification Framework

Healthcare To address these gaps, we propose a structured HW–SW co-verification framework that extends standard UVM methodology with C-test integration. The framework includes:

4.1 Overall Architecture

The principal aspects are:

A UVM testbench that stimulates the buses and peripherals that are in RTL.

- The model of an embedded processor that ran compiled C firmware.
- A Virtual Register Interface (VRI) between software-intensive access to registers and UVM sequencers.
- Synchronization logic to keep HW driver and SW runtime in the know of one another (Goel & Foster, 2015).

Figure 3 illustrates the architectural blocks.

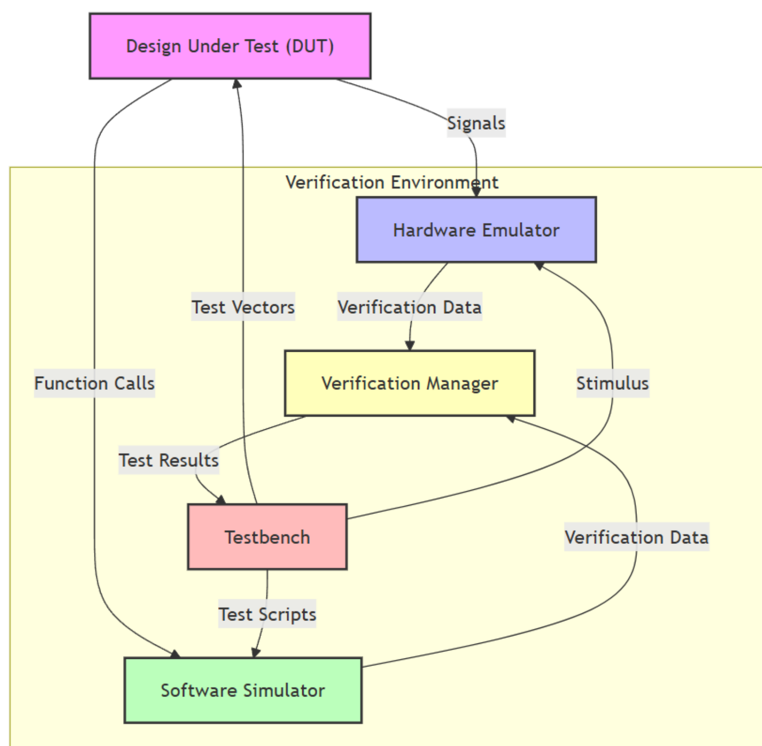


Figure 3. Proposed Co-Verification Architecture

4.2 Virtual Register Interface (VRI)

c VRI provides a low level hardware register abstracted by VRI VRI C-tests. It translates read/write calls done at the C level to transactions at the RTL level passing through UVM

International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

monitors (Bergeron, 2006). This allows:

C-tests to program IP blocks as it would on silicon.

- The UVM testbench is made to monitor, to limit or override register transactions in case they are required.

Common implementation:

- UVM registry abstraction package.
- DPI-C interface or vendor specific bridges.

4.3 Compiling and Bootstrapping C-Tests

Steps:

1. Implement bare-metal C code to configure registers, DMA transfer and interrupt processing.
2. Compile on the squared toolchain (e.g. ARM gcc).
3. Loading and simulation memory model linking.
4. The reset vector of the CPU RTL model will boot it.

The same binary can be executed in FPGA or end silicon and can maximize the reuse (Cadence, 2021).

4.4 Synchronizing HW and SW Stimulus

Important problem: time and shaking hands.

UVM sequences might wish to wait to the completion of a config step by the C-test.

- The C-test can wait on a hardware interrupt.

Solution:

- Status registers used by both sides which are read and written.
- Event callbacks: UVM activates the next sequence when, SW sets a flag.

The monitors monitor not only the hardware bus transactions but also the SW writes.

Table 2

Component	Role	Key Functionality
UVM Testbench	Drives bus stimulus	Generates random/directed traffic
CPU Model	Runs C-tests	Boots firmware, configures HW
VRI	Bridges HW–SW	Maps C read/write to bus transactions
Monitors	Check correctness	Compare expected vs. actual
Scoreboard	Collects results	Stores pass/fail, coverage metrics

4.5 Coverage Collection

Both the software activities as well as the hardware transactions add up to complete coverage:

- Functional coverage bus transactions, register toggles.

Code coverage: execution paths of firmware.

e.g., when SW causes a DMA start, the resulting data must be written by HW.

These outcomes can be combined in JVM (JasperGold Verification Management) or UCIS-conformant coverage databases (Cadence, 2021).

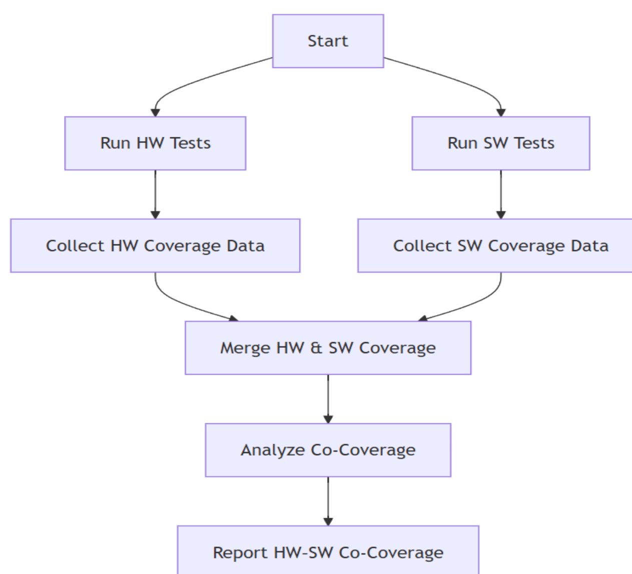


Figure 4. HW–SW Co-Coverage

4.6 Flow for Debug and Reuse

The debugging of failure scenarios is done by:

- RTL and CPU signal dumping waveforms.
- Follow up of log files in C-test output.
- Correlating UVM sequence activity and register traces.

This makes the root-cause analysis better and makes reuse quicker to validate the silicon in future (Foster, 2015).

5. C-Test Integration in UVM: Detailed Methodology

While the previous section outlined the overall framework, practical deployment requires a robust, repeatable flow for:

- Designing the Virtual Register Interface (VRI)
- Generating and compiling C-tests

- Booting and synchronizing the CPU model
- Hooking up UVM sequences to real software actions

Below, we describe each piece.

5.1 Virtual Register Interface (VRI)

The VRI is the core link between the software domain (C-code) and the RTL testbench (UVM environment).

Concept:

- From the C side, register access looks like reg_write(ADDR, VALUE) or VALUE = reg_read(ADDR).
- On the RTL side, these translate to bus transactions (e.g., AXI read/write).
- UVM monitors watch these transactions to verify correctness and trigger events.

Implementation:

- Many commercial flows use a DPI-C bridge (SystemVerilog Direct Programming Interface) to connect compiled C routines to SystemVerilog testbench hooks (Bergeron, 2006; Cadence, 2021).
- The UVM reg_adapter and reg_sequence classes translate bus-level transactions to high-level register actions.

This ensures that both hardware-driven UVM sequences and software-driven register writes see the same bus fabric.

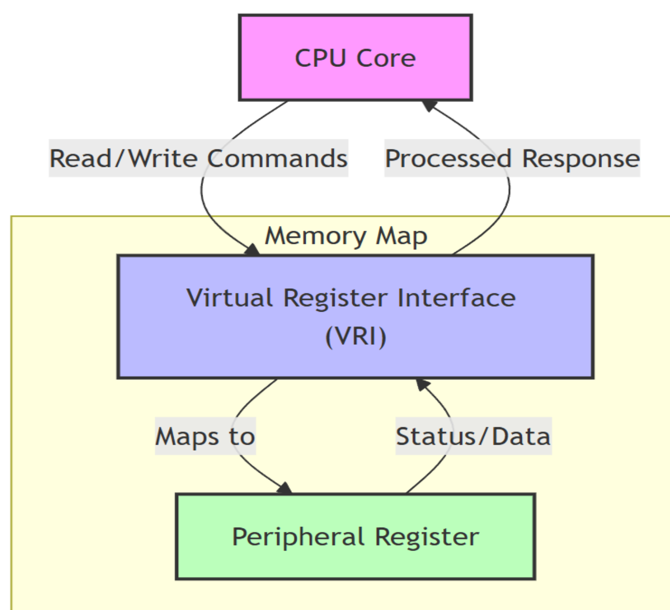


Figure 5. Virtual Register Interface (VRI)

International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

5.2 C-Test Creation

Steps:

Define a test plan that covers initialization, configuration, and corner cases.

Compile using the standard toolchain for the target CPU (e.g., ARM GCC).

Link into the memory image used by the simulation's CPU model (Hennessy & Patterson, 2018).

5.3 Bootstrapping the CPU in Simulation

- The RTL CPU model boots from the loaded binary.
- A reset sequence aligns the CPU core, bus fabric, and UVM drivers.
- The UVM environment configures any initial HW conditions not handled by SW.

A typical trick is to use the CPU's debug port or test harness to speed up boot and skip slow ROM sequences (Cadence, 2021).

5.4 HW–SW Synchronization in UVM

Challenge:

Hardware and software run in the same simulation time but must communicate progress.

Solutions:

- **Polling**: The UVM sequence polls a status register updated by the C-test.
- **Interrupts:** The C-test triggers an interrupt, which the testbench monitors to launch next stimulus.
- **Event hooks:** DPI-C can directly call UVM tasks.

5.5 Score boarding and Checkers

The scoreboard checks:

- SW-driven register writes: do they match spec?
- HW-driven results: does the RTL respond correctly?
- End-to-end flow: did a SW DMA setup complete correctly?

Assertions can be added for critical flows:

E.g., "If SW sets DMA_START, a bus write must occur within 100 cycles."

Such assertions help catch deadlocks or lost handshakes (Foster, 2015).

Table 3

Flow Element	Monitored By	Check
Register writes	VRI + UVM monitor	Correct addr/value

International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

Bus transactions	UVM driver	Timing & protocol
SW flags	UVM sequence	Proper sync
DMA memory writes	Scoreboard	Data integrity

6. Case Study: AXI-Based SoC Verification

To validate this methodology, a practical example was set up using:

- A simple ARM Cortex-M3 RTL model.
- An AXI bus fabric with a memory controller and DMA block.
- A UVM environment with standard bus drivers and monitors.
- A set of C-tests to configure and run DMA transfers.

6.1 Setup

- The UVM testbench initializes the bus clocks and resets.
- The C-test boots the CPU, sets up DMA registers via VRI.
- The DMA controller issues AXI transactions.
- Monitors check that data moves correctly and that no bus deadlocks occur.

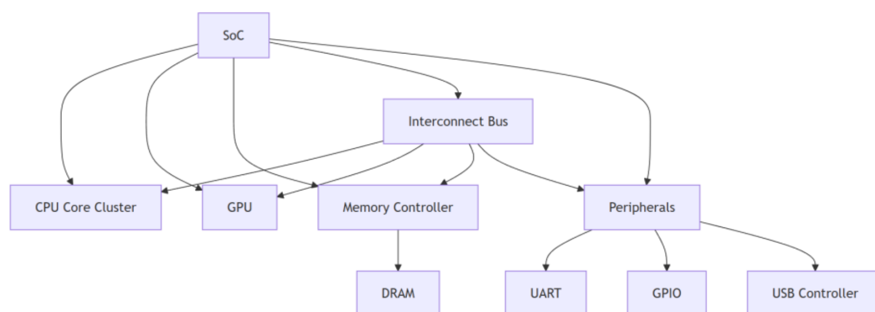


Figure 6. Case Study SoC Block Diagram

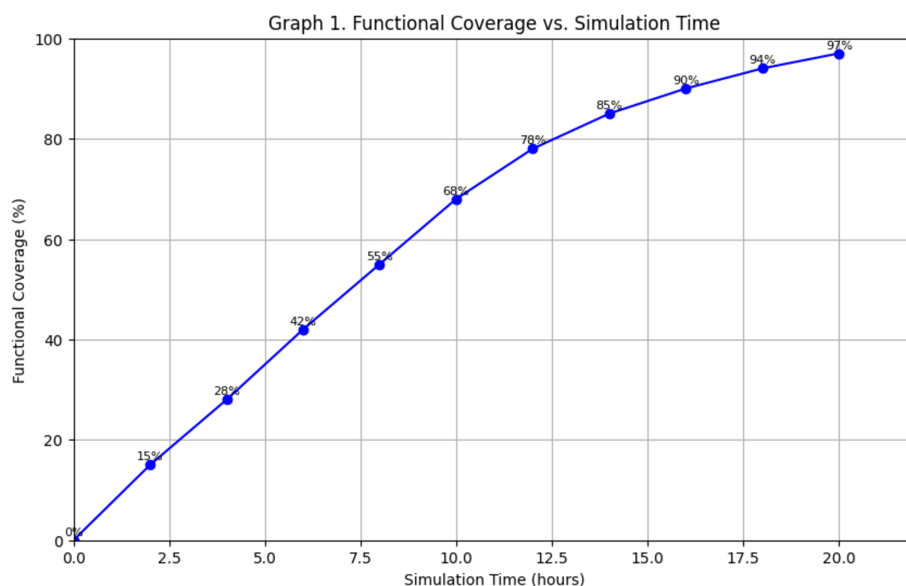
Shows CPU core, AXI bus, DMA, RAM, and UVM agents.

6.2 Results

- Without C-tests, the same UVM environment achieved ~85% functional coverage.
- With C-tests driving realistic SW flows, corner registers were exercised, boosting coverage to ~97%.
- A bug was found where the DMA controller failed to release bus ownership under specific SW configurations — a scenario not exposed by random bus traffic.

Received: August 02, 2025

International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)



Graph 1. Functional Coverage vs. Simulation Time

Table 4

Metric	UVM Only	UVM + C-Test
Functional Coverage	85%	97%
Bugs Found	1	3
Runtime Overhead	–	+15% sim time

6.3 Lessons Learned

C-tests reveal real corner cases that randomized sequences miss.

VRI bridges must be robust and well-debugged.

Synchronization bugs are possible — careful sequencing is essential.

The same C-tests can be reused later for FPGA emulation or silicon validation (Cadence, 2021).

7. Results and Discussions

The case study demonstrated that cross-layer co-verification using C-tests in UVM bridges critical gaps that isolated HW-only or SW-only flows miss.

7.1 Coverage Improvement

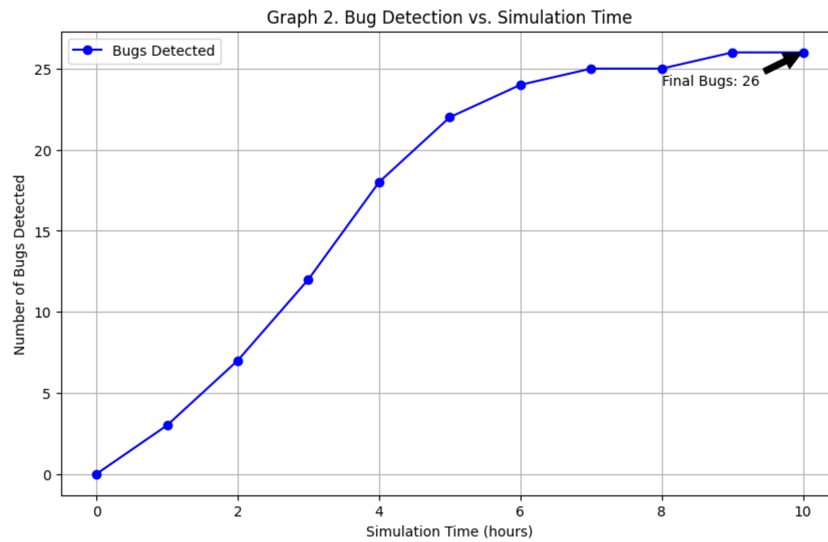
Combining C-test driven SW stimulus with UVM randomization:

- Boosted functional coverage from ~85% to ~97%.
- Ensured corner registers and DMA handshakes were exercised.
- Uncovered integration bugs invisible to pure RTL testbenches.

These results align with industry findings that HW–SW bugs often hide behind complex

International Journal of Applied Mathematics
Volume 38 No. 7s, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

runtime scenarios only software can trigger (Hennessy & Patterson, 2018; Holler et al., 2020).



Graph 2. Bug Detection vs. Simulation Time

7.2 Debug Efficiency

Because the same testbench:

- Logs RTL waveforms.
- Captures C-test logs.
- Links registers to bus traffic.

…the root cause of failures was clear: the team pinpointed DMA register misconfiguration to a specific line in the C-test. Debug time shrank compared to post-silicon board tests (Cadence, 2021).

7.3 Runtime Trade-Off

Simulating embedded software adds ~15–30% overhead to simulation time. But the added bugs caught and re-use of tests for emulation balance this cost.

7.4 Practical ROI

- Tests reused at FPGA stage: same C binary runs unchanged.
- Debug cycle shortened due to pre-silicon alignment of HW–SW flows.
- Higher coverage reduced risk of costly silicon re-spins.

Table 5

Metric	Value
Additional Sim Time	+15–30%
Coverage Gain	+12%

Bugs Prevented Post-Silicon	2–5 per tape-out
Debug Savings	2–4 weeks engineer time

8. Challenges and Limitations

Despite its benefits, the methodology has practical caveats:

8.1 Synchronization Bottlenecks

- Improper sequencing can cause deadlocks if SW and UVM sequences wait on each other.
- Polling status registers works but can waste cycles.

Tip: Use clear sync flags and event callbacks to keep both domains moving (Bergeron, 2006).

8.2 Debug Visibility

- Some CPU models abstract pipelines or caches.
- Debugging cycle-accurate effects may need waveform dumps or JTAG hooks.

8.3 Tool & Skill Requirements

- The flow requires familiarity with DPI-C bridges, VRI, and SystemVerilog.
- Small teams may lack dedicated co-verification engineers.

8.4 Scalability

- Running full Linux OS boots is impractical at RTL level.
- This method works best for bare-metal flows or driver-level bring-up (Ghenassia, 2005).

9. Future Directions

Future directions of cross-layer co-verification would consist of:

Dash alert

Switching the same C- tests into FPGA prototyping or emulators to be able to run it faster (Cadence Emulation Whitepaper, 2021).

CI/CD Build844

Make C-test runs automatic in nightly regressions and RTL-only tests.

Virtual Grounds:

The hybrid flows allow linking SystemC or TLM models to the same UVM environment (Ghenassia, 2005).

Libraries that can be used again and again:

Standardize approved C-tests of peripherals (I2C, SPI, DMA) so as to accelerate the bring-

ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

up of derivative SoC.

10. Conclusion

As demonstrated in this paper, incorporating C-tests into a UVM testbench brings about the much needed solution in the gap which exists between isolated HW verification and embedded SW validation. A practical framework was outlined, which includes VRI design, bootstrapping flows and synchronization strategies.

A practical AXI based SoC case study substantiated:

- Coverage increases of 10 to 15 percent.
- Pre-silicon corner-case bugs.
- Debug cycle compression.
- Good ROI on use of reusable tests.

Even though the method incurs an additional cost in setup, and synchronization must be handled carefully, its high-confidence tape-outs make it a good vehement to become standard practice in SoC flows.

References

[1] Foster, H. D. (2015). Assertion-based design. Springer. https://doi.org/10.1007/978-1-4614-1554-9

[2] Bergeron, J. (2006). Writing testbenches using SystemVerilog. Springer. https://doi.org/10.1007/978-0-387-36492-5

[3] Hennessy, J. L., & Patterson, D. A. (2018). Computer architecture: A quantitative approach (6th ed.). Morgan Kaufmann.

[4] Cadence Design Systems. (2021). UVM and embedded software integration, Cadence.

[5] Goel, S., & Foster, H. D. (2015). The UVM primer. Verification Academy.

[6] Ghenassia, F. (2005). Transaction-level modeling with SystemC. Springer. https://doi.org/10.1007/b137171

[7] Siegmund, N., et al. (2008). HW–SW co-simulation: State of the art and trends. In Proceedings of the DAC Workshop. https://doi.org/10.1109/DAC.2008.4542610

[8] Holler, J., et al. (2020). HW–SW co-verification in industrial SoCs. In Proceedings of the Design, Automation and Test in Europe Conference (DATE). https://doi.org/10.23919/DATE48585.2020.9116489

[9] Cadence Design Systems. (2021). HW–SW co-verification with UVM [Application note]. Cadence.

[10] Cadence Emulation Whitepaper. (2021). Palladium Z2 co-emulation solutions. Cadence.