I2CMB Verification Progression - From Class Hierarchy to Closed Coverage

Class-based SystemVerilog · ncsu_pkg UVM-style framework · 20-item test plan · QuestaSim 2024.2 regression with UCDB merge · NCSU ECE 745 (ASIC Verification) · Spring 2026 · Projects 2-4

Topics: *I²C Multi-Bus Controller · VHDL DUT in SystemVerilog bench · Wishbone + dual I²C agent packages · ncsu_pkg config DB hand-off · 3 covergroups (wb / i2c / fsm) · 20-item test plan in XML · directed + constrained-random stimulus · QuestaSim regression flow · xml2ucdb merge · 95.24% closure*

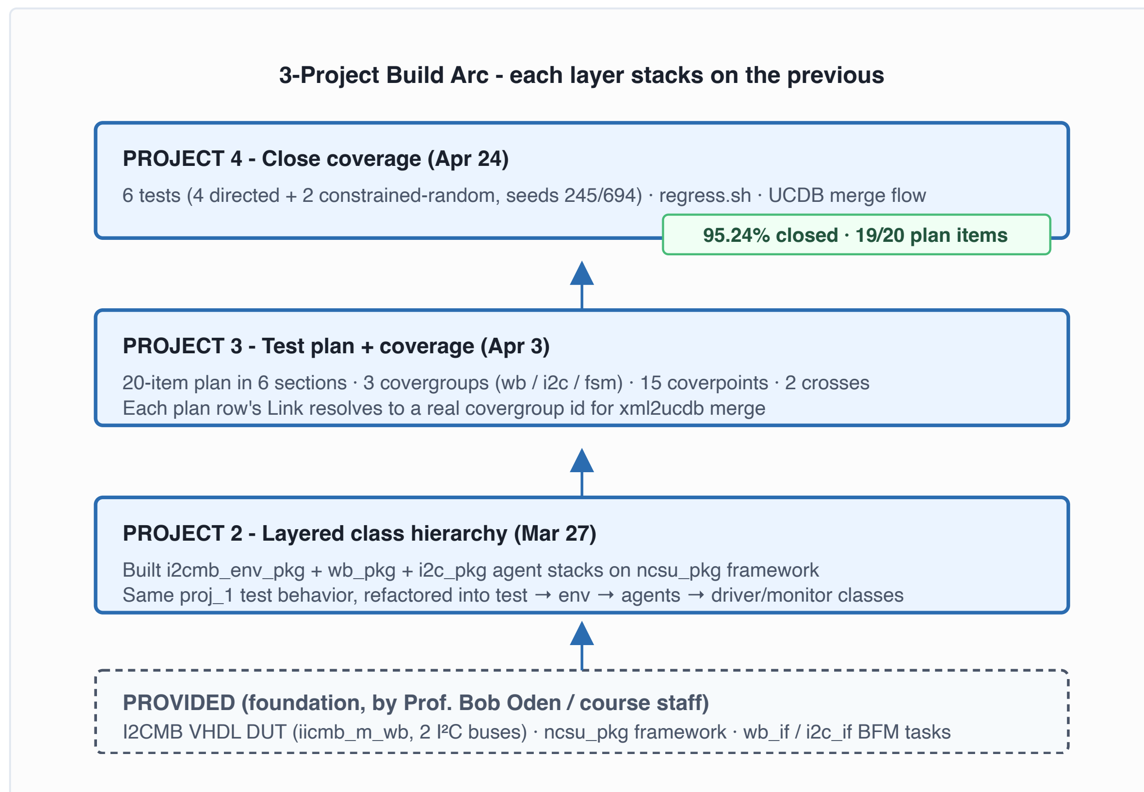
> **The one-liner:** ECE 745 builds an I2CMB verification environment in three project phases. P2 refactors a flat proj_1 test into a layered class hierarchy on the ncsu_pkg framework (Wishbone + 2× I²C agents, env, generator, scoreboard, predictor). P3 writes a 20-item test plan in Excel/XML and implements three covergroups (wb / i2c / fsm) that link to it. P4 closes coverage with four directed and two constrained-random tests, builds a regress.sh that merges per-test UCDBs with the test-plan UCDB, and lands at 95.24% overall coverage with 19/20 test-plan items at 100%.
>
> **Why it's interesting:** This is the build-arc story, not the final-state architecture. It shows the methodology - turning an unstructured Lab-1-style bench into a UVM-shaped, plan-driven, coverage-closed regression. Worth noting: ncsu_pkg is not real UVM, and the scoreboard's predictor is stubbed, so coverage closure is the real signal, not byte-for-byte data checking.

The 3-project arc - what each phase delivered

The course runs four projects on the same I2CMB DUT. P1 was a flat top.sv that wrote raw Wishbone tasks. Projects 2-4 take that working bench and convert it, in three discrete deliverables, into a real verification environment.

The figure below shows the stacked-layer view. Each project's deliverables are summarized inline; full details for the covergroups and regression flow are in the dedicated sections below.

This arc mirrors how a verification environment matures in practice - going from "I can write a directed test" to "I can read a spec, derive a plan, implement covergroups that mechanically prove the plan, and close coverage on a real DUT." The three-project sequence walks through that progression in order, each phase delivering a concrete verification artifact.



Methodology view: P2 turned the unstructured proj_1 testbench into a class hierarchy. P3 added the test plan + 3 covergroups that mechanically link to plan rows. P4 closed coverage to 95.24% with 6 tests + regression flow. Dashed bottom block = course-provided foundation; blue blocks = our deliverables.

The DUT - I2CMB in one paragraph

The I²C Multi-Bus Controller is an open-source VHDL design (Sergey Shuvalkin, OpenCores) - a Wishbone-slave wrapper around an I²C master. The Wishbone side exposes four byte-wide registers at `adr[1:0]`: `CSR` (control/status, with enable + IRQ-enable bits), `DPR` (data/parameter), `CMDR` (command, with the DON completion bit + 3-bit opcode), `FSMR` (FSM-state read-only). The I²C side fans out to **NUM_I2C_BUSSES = 2** independent buses (SCL[0:1], SDA[0:1]) with a per-bus mux selected by the `CMD_SET_BUS` command. Internally the DUT runs a byte-level FSM (BFSM) with 8 named states (IDLE / BUS_IS_TAKEN / START_IS_PENDING / START / STOP / WRITE_BYTE / READ_BYTE / WAIT) - those are the states the software sees through FSMR and the ones the testbench covers - plus a bit-level FSM (BIT_FSM) underneath that handles SCL/SDA toggling and ACK/NAK, which is not directly observable.

The top.sv hookup is mixed-language: VHDL DUT compiled via `vcom`, SystemVerilog bench via `vlog`. The instantiation uses Questa's mixed-language syntax: `\work.iicmb_m_wb(str) #(.g_bus_num(NUM_I2C_BUSSES)) DUT(...)`. The bench owns clock generation (10 ns period), reset generation (active high, deasserted at 113 ns), the Wishbone master BFM, two I²C BFM instances, and the ncsu_config_db hand-off that gives the env class hierarchy its virtual interface handles.

Agents + environment - how the class hierarchy is wired

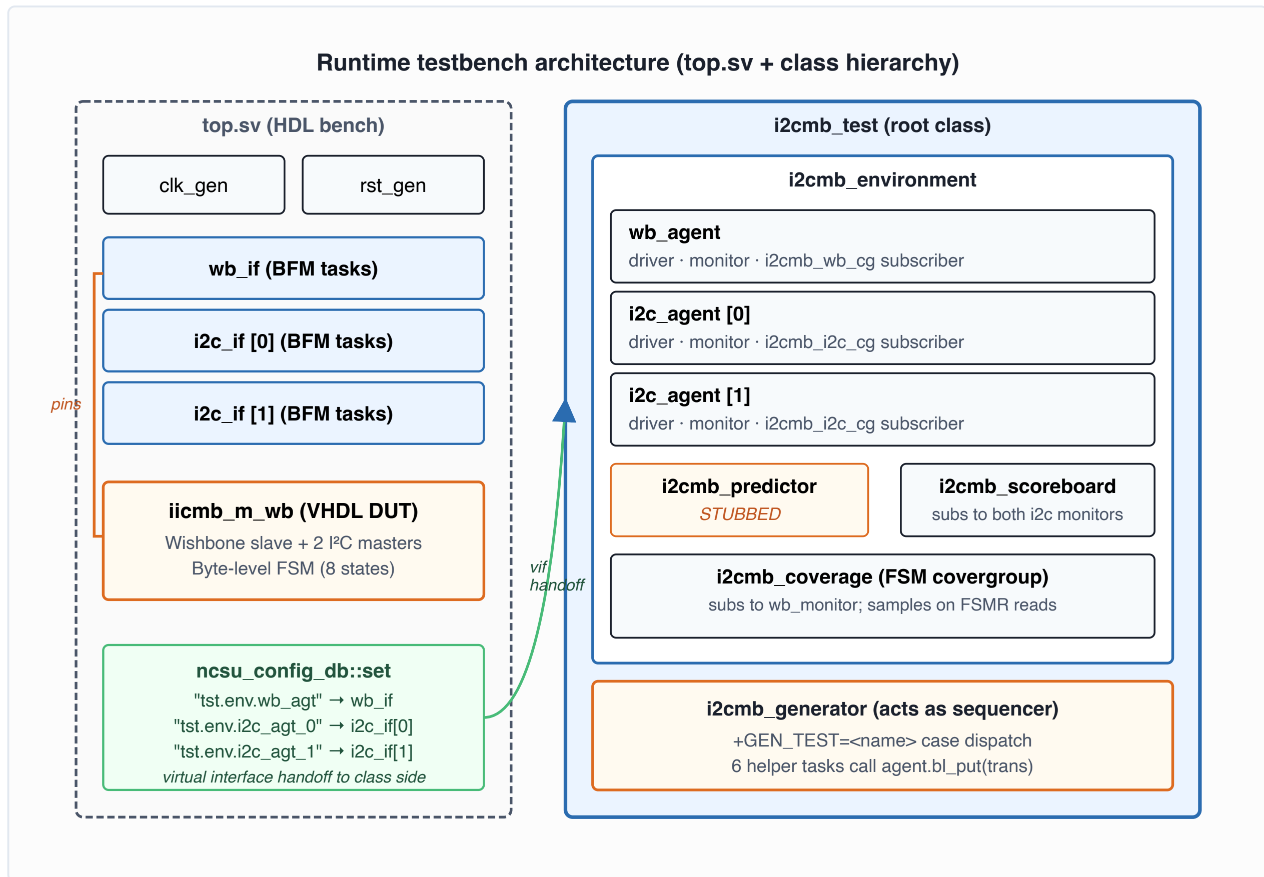
The runtime hierarchy (built top-down in P2):

```
top.sv (module – clk/rst, BFMs, DUT, ncsu_config_db handoff)
└─ i2cmb_test                (root class – constructs env, generator)
   ├─ i2cmb_environment      (builds + connects all components)
   │  ├─ wb_agent             (driver + monitor + i2cmb_wb_cg subscriber)
   │  ├─ i2c_agent[0]         (driver + monitor + i2cmb_i2c_cg subscriber)
   │  ├─ i2c_agent[1]         (driver + monitor + i2cmb_i2c_cg subscriber)
   │  ├─ i2cmb_predictor      (subscribes to wb_monitor; sets scoreboard expected_trans – STUBBED)
   │  ├─ i2cmb_scoreboard     (subscribes to both i2c_monitors; compares actual vs expected)
   │  └─ i2cmb_coverage       (subscribes to wb_monitor; samples i2cmb_fsm_cg on FSMR reads)
   └─ i2cmb_generator         (acts as the sequencer – dispatches +GEN_TEST=<name> via case)
```

Three integration details worth knowing:

1. **Virtual interfaces cross the module/class boundary via `ncsu_config_db`.** Exact UVM analog. top.sv calls `ncsu_config_db#(virtual wb_if)::set("tst.env.wb_agt", wb_bus)` (and similar for the two i2c_if handles) *before* constructing the test class. Each agent's constructor calls `::get(get_full_name(), this.bus)` - so the full hierarchical name at construction time must match what top.sv set. An off-by-one in the path fails silently with a null vif that doesn't crash until the first BFM call.
2. **Monitor → subscriber via a hand-rolled analysis port.** Each agent owns `ncsu_component#(T) subscribers[$]`. `connect_subscriber(s)` pushes onto the queue, and the agent's `nb_put(t)` loops over the queue and calls each subscriber's `nb_put(t)`. Identical pattern to UVM's analysis port + subscribers; no formal TLM port machinery.
3. **The generator IS the sequencer.** There is no separate `wb_sequencer` or `i2c_sequencer` class. The generator's helper tasks (`wb_write`, `wb_read`, `i2c_start`, etc.) construct transaction objects and call `agent.bl_put(trans)` directly, which forwards to `driver.bl_put`. Less idiomatic than UVM's separate sequencer, but it's what the course flow uses and what `regress.sh` invokes via `+GEN_TEST`.

Each agent contains a `*_configuration` (knobs), a `*_transaction` (data class on the wire - i2c_transaction has `rand` addr/op/data with constraints), a `*_driver` (calls BFM tasks via `bl_put`), a `*_monitor` (decodes bus activity and broadcasts via `typed_parent.nb_put`), a `*_coverage` (subscriber sampling the covergroup), and a `*_agent` container that calls `ncsu_config_db::get` for its vif and fails loudly with `ncsu_fatal` if the hierarchy path doesn't resolve. The `*_if.sv` interface holds the BFM tasks (`master_write/read`, `wait_for_interrupt/reset` on Wishbone side; `wait_for_i2c_transfer`, `provide_read_data`, `monitor` on I²C side).



Runtime view: HDL side (top.sv, dashed gray box) instantiates the BFMs and the VHDL DUT, then hands off virtual interfaces to the class hierarchy via ncsu_config_db. SV side (i2cmb_test, blue) builds env containing 3 agents + predictor (stubbed, orange) + scoreboard + coverage (FSM cg), plus the generator which dispatches one of 6 tests via the +GEN_TEST plusarg.

The IRQ handshake in `wb_wait_for_irq` is the trick that makes FSM coverage work: every I2CMB command completes asynchronously with an interrupt, so the helper forks a polling branch that reads FSMR every 100 ns to expose intermediate states to the coverage sampler, alongside a blocking branch on `bus.wait_for_interrupt()`. Without the polling, the FSM walks through START_IS_PENDING / START / BUS_IS_TAKEN / WRITE_BYTE between IRQs and the `fsm_state` coverpoint never sees those states. "Active scoreboarding via observation" - the test deliberately makes the DUT observable.

Coverage model - three covergroups, 15 coverpoints, 2 crosses

The functional-coverage model is split across three covergroups, one per interface or environment package. Each covergroup lives next to the monitor that samples it, so the agents stay reusable on a different DUT and the UCDB browser shows three independent coverage trees (wb / i2c / env) rather than one flat mixed-trigger group.

Covergroup	File	Sampling trigger	Coverpoints + crosses
`i2cmb_wb_cg`	`wb_pkg/src/wb_coverage.svh`	Every observed wb_transaction (CSR/DPR/CMDR writes + CMDR/FSMR reads).	8 coverpoints: `csr_e_bit`, `csr_ie_bit`, `dpr_data` (min/mid/max bins), `cmdr_don`, `cmdr_cmd` (all 7 opcodes), `irq_assert`, `irq_clear`, `set_bus_cmd`. Cross: `multi_bus_r_w = set_bus_cmd × cmdr_cmd` with `ignore_bins` filtering out wait/start/stop/set_bus combinations.
`i2cmb_i2c_cg`	`i2c_pkg/src/i2c_coverage.svh`	Every observed i2c_transaction, sampled per data byte (so a 4-byte burst samples 4 times).	4 coverpoints: `i2c_op` (R/W), `i2c_read_op` (R/W variant), `i2c_addr` (min/mid/max bins), `i2c_data` (min/mid/max bins). Cross: `i2c_comb = i2c_op × i2c_addr`.
`i2cmb_fsm_cg`	`env_pkg/src/i2cmb_coverage.svh`	Wishbone read of FSMR specifically (filtered subset of wb_transaction stream).	3 coverpoints: `fsm_state` (8 bins, one per BFSM state), `fsm_trans` (5 bins for the start-of-transfer arc: START_IS_PENDING → START → BUS_IS_TAKEN → WRITE_BYTE/READ_BYTE), `fsm_idle` (3 bins for the end-of-transfer arc: BUS_IS_TAKEN → STOP → IDLE).

The FSM transition tracking is the subtle bit: `fsm_trans` and `fsm_idle` are plain coverpoints, but they're only written when a procedural guard confirms that `prev_fsm_state → fsm_state` matches an expected arc. So a

covered bin means "I observed THIS transition," not just "I observed this state in isolation." Transition coverage expressed via prev-state guards rather than SystemVerilog's native `bins x => y` syntax - cleaner to read in the UCDB report.

The 20-item test plan (P3 deliverable) is structured into six sections - Register Testing (1.1-1.5), FSM Flow Testing (2.1-2.4), I²C Operations (3.1-3.6), Interrupt Behavior (4.1-4.2), Multi-Bus Selection (5.1-5.2), and Code Coverage (6.1, `/top/DUT` instance). Each row's Link column resolves to a real covergroup / coverpoint / cross identifier (e.g., row 1.1 Link = `i2cmb_wb_cg::csr_e_bit`) that matches the covergroup name + coverpoint name in source. When `xml2ucdb` converts the plan and `vcover merge` joins it with the simulation UCDB, the Questa viewer shows test-plan rows annotated with their covergroup hit rates side-by-side. If a Link doesn't resolve, the plan row stays at 0% even when the underlying covergroup hits 100%, so the naming has to be exact.

Regression flow - 6 tests, UCDB merge, 95.24% closure

The six tests (P4 deliverable) are dispatched from a single compiled testbench image via `+GEN_TEST=<name>`. The generator's `run` task case-statements on the plusarg:

Test	Seed	Style	What it closes
`directed_test_bus_0`	1	Directed	Boundary writes + reads on bus 0 (min/max addr, min/max data, single + 2-byte read). Hits wb start-arc + stop-arc bins and i2c addr/data min+max bins.
`directed_test_bus_1`	1	Directed	Same arc on bus 1. Closes the `set_bus_cmd::bus_1` bin and the bus 1 side of the `multi_bus_r_w` cross.
`directed_test_csr`	1	Directed	Toggles CSR enable + IRQ-enable. Closes `csr_e_bit::disabled` and `csr_ie_bit::disabled` (the happy-path tests never disable the core).
`directed_test_wait`	1	Directed	Issues `CMD_WAIT`. Closes the `FSM_WAIT` bin in `fsm_state` - the one state random transfers don't reach.
`random_test_bus_0`	245	Constrained-random	10 randomized transactions on bus 0. Hits mid-range addr/data bins and a wider mix of R/W combinations.
`random_test_bus_1`	694	Constrained-random	Same on bus 1. Seeds 245 / 694 were selected to hit FSM-transition bins the four directed tests don't reliably cover.

The full flow runs via `regress.sh`, which iterates over the `testlist` file (test name + seed per line), calls `make run_test TEST= SEED=` for each, then calls `make run_proj4` for the merge:

```
# per-test run (sim/Makefile)
qrun -64 -cover sbceft -coverage -onfinish stop \
  +GEN_TEST=$(TEST) -sv_seed $(SEED) \
  -do "coverage save -onexit test_$(TEST).ucdb; run -all; exit"

# merge (sim/Makefile run_proj4 target)
vcover merge -out merged_tests.ucdb     test_*.ucdb
xml2ucdb -format Excel i2cmb_test_plan.xml i2cmb_test_plan.ucdb
vcover merge -out regression.ucdb       i2cmb_test_plan.ucdb merged_tests.ucdb
vsim -viewcov regression.ucdb
```

Coverage closure landed at **95.24%** overall, with **19/20 test-plan items at 100%**. The one unclosed item is `6.1 /top/DUT` - the instance-level code-coverage row asking for 100% statement + branch on the DUT. Hitting 100% on RTL statement+branch requires per-state stimulus that this 6-test plan doesn't generate (specifically the error / arbitration-lost / NAK paths in the bit-level FSM, which are not driven by any happy-path test). Closing it would require adding tests that deliberately corrupt the I²C bus during a transfer.

Design decisions and trade-offs

Three limitations of the testbench, and the reasoning behind each.

1. **Predictor is stubbed; the scoreboard cannot actually fail.** `i2cmb_predictor::nb_put` has its `scoreboard.nb_transport(trans, predicted_trans)` call commented out - so `expected_trans` is never set. The scoreboard's compare-and-log path runs, but with `expected_trans == null` it always falls through. Coverage closure is the real verification signal here; byte-for-byte data checking is not happening. A real DV deliverable would implement the predictor: parse the wb command stream (`CMD_START` , `CMD_WRITE` , `CMD_READ_WITH_ACK/NAK` , `CMD_STOP`) and reconstruct the predicted i2c_transaction.
2. **ncsu_pkg is not real UVM.** The architecture maps cleanly - `ncsu_component_base` ↔ `uvm_component` , `ncsu_transaction` ↔ `uvm_sequence_item` , `ncsu_config_db` ↔ `uvm_config_db` , subscriber queue ↔ analysis port - but ncsu_pkg has no UVM phasing FSM, no objections, no factory overrides, no separate sequencer class, no formal TLM port hierarchy. The patterns transfer, but this is not the real UVM framework.
3. **The 95.24% / 19-of-20 number is from the regression UCDB but I don't have a screenshot committed to source control.** The methodology - `regress.sh` , `vcover merge` , `xml2ucdb` - is reproducible from the submitted code, but the specific closure summary lives in QuestaSim at view-time. The unclosed item is `6.1 /top/DUT` code-coverage instance; that's the structural holdout given a coverage plan built around protocol behavior rather than RTL statement reachability.

Common questions about this project

Question	Short answer
How does a transaction flow from generator to DUT and back?	Generator constructs a wb_transaction (or i2c_transaction) and calls `agent.bl_put(trans)`. Agent forwards to `driver.bl_put`, which calls the BFM tasks in the SV interface (`master_write` / `master_read` for Wishbone). DUT activity is observed by the monitor, which builds an observed transaction and calls `typed_parent.nb_put`. The agent loops over `subscribers[$]` and forwards to coverage / predictor / scoreboard.
Why three covergroups instead of one big one?	Each covergroup lives in the interface package that owns its sample trigger. wb_cg samples on Wishbone monitor events, i2c_cg samples per I²C byte, fsm_cg samples on FSMR reads specifically. Co-locating covergroup with monitor keeps agents reusable on a different DUT and keeps the UCDB hierarchy mirroring the testbench - coverage browser shows three independent trees.
Two I²C agents, or one parameterized with bus_id?	Two instances of the same parameterized class, each owning one virtual interface. The DUT has two physical buses wired to two `i2c_if` instances in top.sv; each agent gets one via `ncsu_config_db`. The `bus_id` parameter in generator helper tasks selects WHICH agent to call `bl_put` on. Sharing one agent across two buses would require muxing in the driver - avoided.
How is randomization constrained?	`i2c_transaction` has `rand bit[6:0] addr`, `rand i2c_op_t op`, `rand bit[7:0] data[]` with constraints: addr in [0:7F], data[i] in [0:FF], data.size() in [1:4]. So one randomization yields up to a 4-byte burst with a valid I²C address. Generator calls `trans.randomize()` and dispatches based on `trans.op`.
Why does `directed_test_csr` exist if it doesn't drive I²C?	Coverage closure. It disables then re-enables the core - the only way to hit `csr_e_bit::disabled` and `csr_ie_bit::disabled` bins, since every other test leaves the core enabled. Hitting both values of a Boolean coverpoint requires deliberately covering both.
How would full self-checking be added?	Implement `i2cmb_predictor::nb_put`: parse the wb command stream (CMD_START → CMD_WRITE/CMD_READ_* → CMD_STOP), reconstruct the predicted i2c_transaction, call `scoreboard.nb_transport(predicted, _)` to set `expected_trans`. Scoreboard's compare path is already wired; the missing piece is the predictor logic.

Question	Short answer
What was the hardest part?	Getting the virtual-interface hand-off right via ncsu_config_db. The agent constructor calls `config_db::get(get_full_name(), this.bus)`, which means the FULL HIERARCHICAL NAME at construction time has to match what top.sv set. Off-by-one in the path (e.g., `"tst.env.wb"` vs `"tst.env.wb_agt"`) fails silently with a null vif, which doesn't crash until the first BFM call. Same gotcha exists in real UVM.
What would make this production-grade?	Implement the predictor (highest-value gap - turns coverage-driven verification into self-checking verification). Add SVA on Wishbone (cyc/stb/ack handshake) and I²C (start/stop conditions, ACK bits). Increase random transaction counts from 10 to 100+ per test. Add CSR×FSM cross-coverage and per-bus×command cross-coverage. Add the error / arbitration-lost / NAK paths to close `/top/DUT` code coverage to 100%. Migrate from ncsu_pkg to real UVM if the team standard is UVM.