2025
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Advancing Open-Source Verification: Enabling Full Randomization in Verilator

Yilou Wang,
PlanV GmbH, Munich, Germany

Roadmap

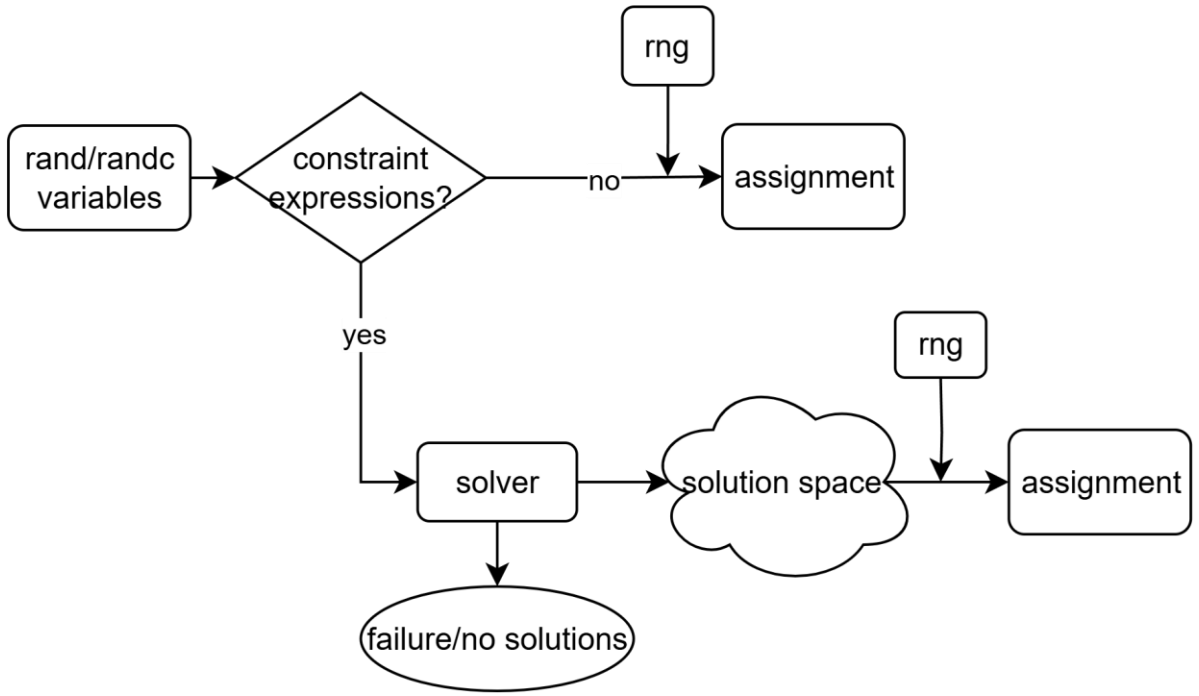
- Background & Motivation
  - Verilator in open-source verification
  - Randomization in SystemVerilog
  - Verilator Before Our Work
- Contribution & Results
  - Basic randomization for aggregates
  - Constrained randomization for aggregates
  - Three UVM testbenches run on Verilator
- Conclusion & Future Work

Open-Source Verification on the Rise

- Open-source EDA is receiving increasing attention and adoption
- Verilator: the most widely used open-source RTL simulator
  - Fast, Cycle-accurate, Open-source
- But limited SystemVerilog feature support, especially for advanced verification features like **randomization**
- This gap prevents Verilator from being more widely used in UVM-based verification

Randomization in SystemVerilog

- Core to functional verification → generates diverse stimuli
- Two main forms:
  - Basic randomization → pure random values
  - Constrained randomization → random values under user-defined rules



Verilator's Existing Randomization (Before Our Work)

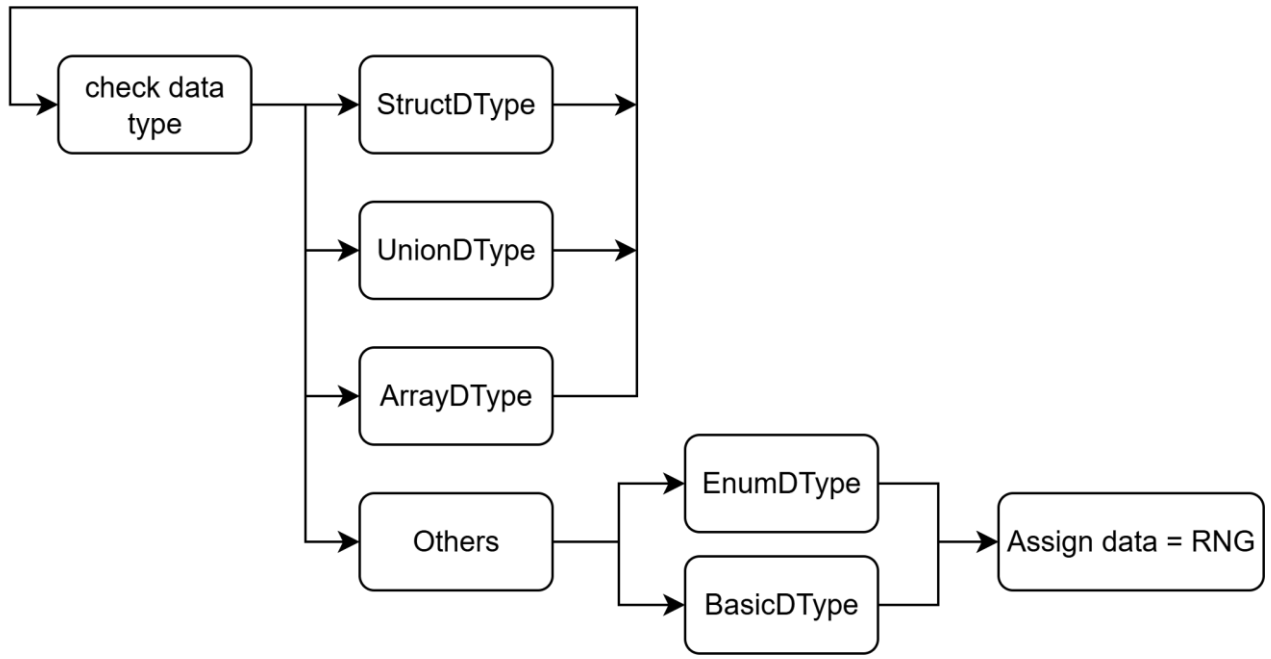
- Basic randomization (primitive only)
  - Internal RNG generates value → masked to bit width → assigned to var
- Constrained randomization (primitive only)
  - External SMT-LIB2 solver integration
  - Verilator translates constraint blocks → solver expressions (smt-lib2 format)
- Incomplete Randomization support → UVM testbenches cannot run
- Our goal is to **support full randomization in Verilator**, moving it closer to advanced open-source verification capabilities.

Our Contribution

- Extended Verilator randomization beyond primitives
- Now supports:
  - Arrays (fixed, dynamic, queue, associative)
  - Structs (packed & unpacked)
  - Unions
  - Nested combinations
- Both basic and constrained.

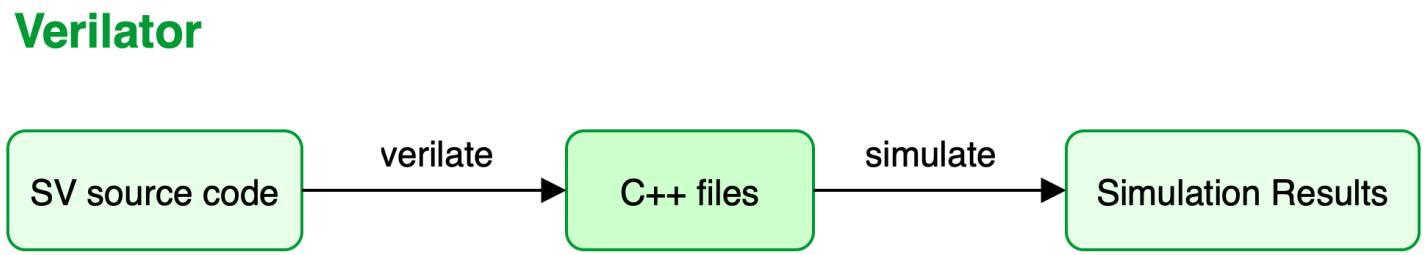
Basic Randomization for Aggregates

- Aggregated types = composite (structs, arrays, unions)
- **Recursive traversal** → peel layer by layer
- Stop at primitive type → apply existing RNG support



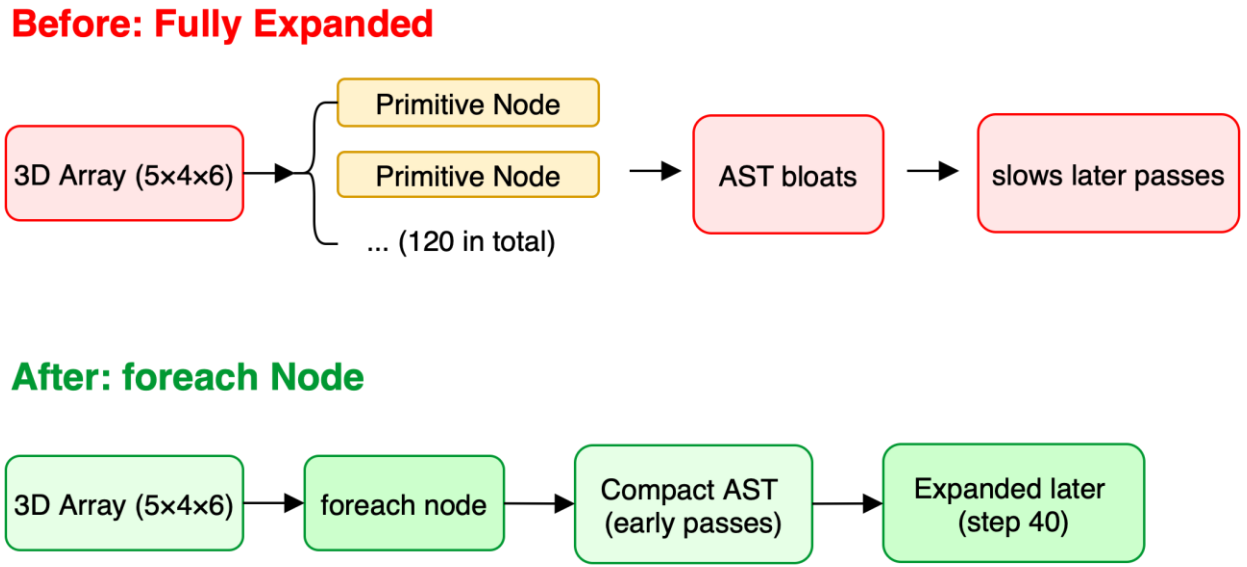
The Challenge: AST Explosion

- Example: 3D array (5×4×6) → 120 primitive elements
- In Verilate phase: SV → AST → C++
- Randomization handled early (e.g., step 13)
- If fully unrolled → AST becomes very large
- Large AST propagates through all later passes → hurts performance



Solution: foreach Node

- Utilize foreach node in AST as loop abstraction for arrays
- Expansion delayed to later step (e.g., step 40)
- Improves performance without losing functionality



Constrained Randomization for Aggregates

- Existing flow:
  - Verilate phase → translate constraints into SMT-LIB2 format
  - Simulate phase → call external solver to find solutions
- Extension to aggregates: same core idea → decompose into primitives
- Different decomposition strategies:
  - Arrays → flatten into elements (index-based)
  - Structs/Unions → flatten into members (dot notation)
  - Nested → combine both

Constrained Randomization for Arrays

- Arrays
  - packed array → treated as a long vector
  - unpacked array
    - fixed-size → fixed size of index, like `arr[2][3]`
    - variable-size
      - dynamic array/queue → like `arr[ ]`
      - associative array → index can be everything, like `arr[string]`

mapped to SMT-LIB2 array logic (QF_ABV)

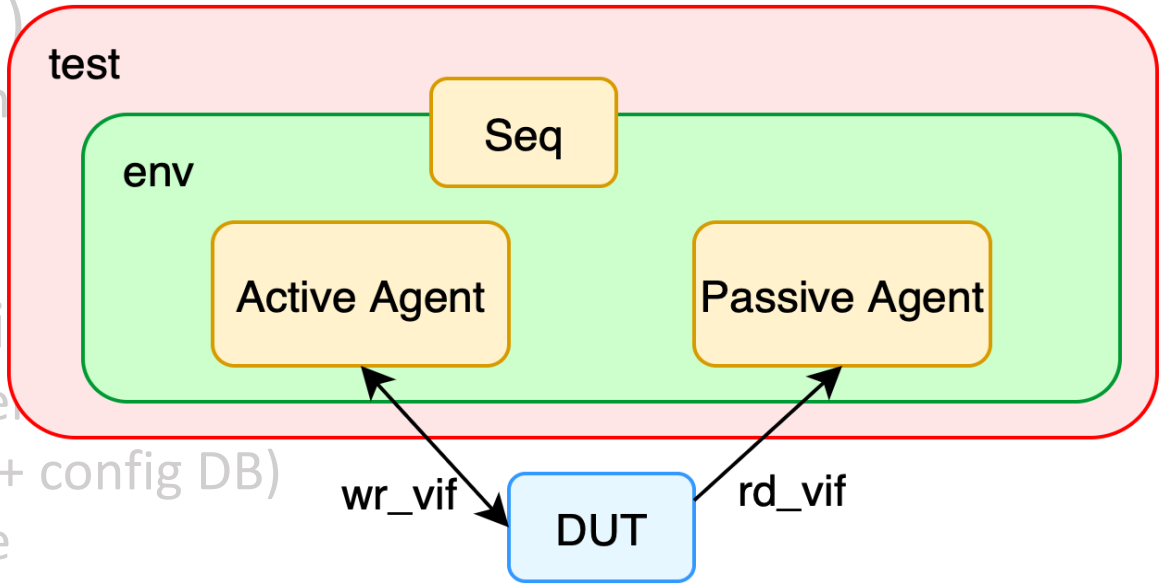
- In QF_ABV(array-bit-vector)
  - Declare full array as a var
  - Use **select** to fetch the specific elements
  - Use **store** to receive the value of elements
- Enables element-wise constraint solving while preserving array structure

Validation with UVM Testbenches

- Evaluated Verilator with three SV-UVM testbenches
- Different structures & complexity levels
- All validated in QuestaSim (reference)
- Goal → check how far Verilator can go with newly-supported randomization feature

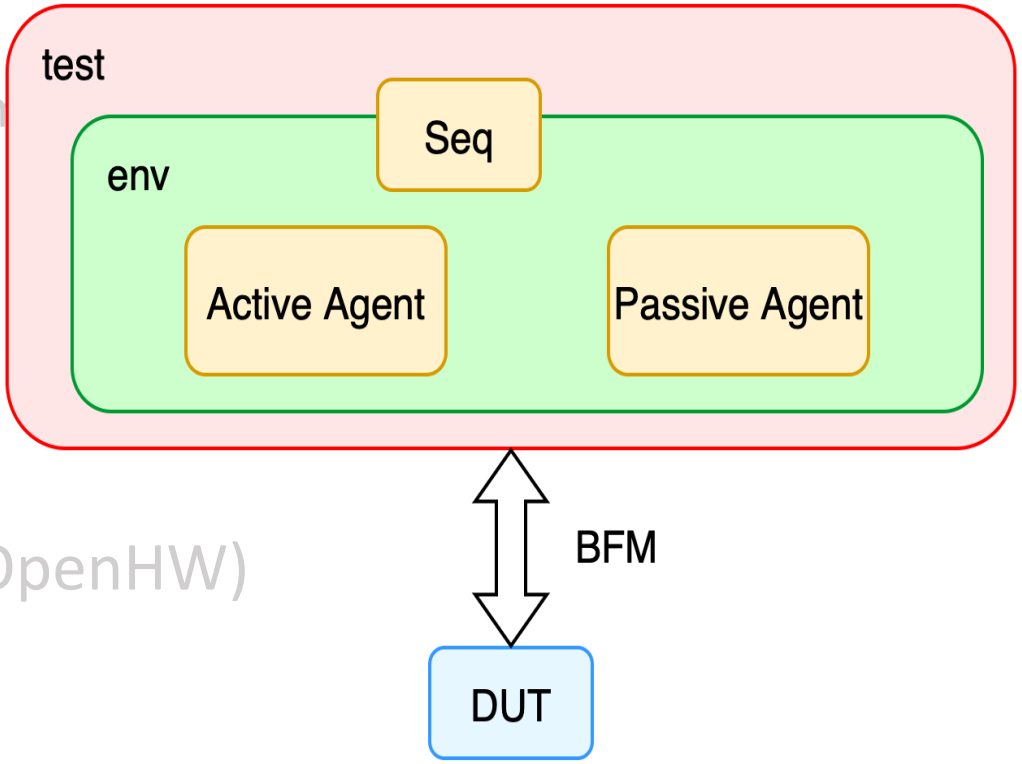
Three UVM Testbenches

- uvm_test_1 (simple agent-based)
  - Active + passive agent
  - Virtual sequence drives constrained random stimulus
  - Result: Pass (Verilate + Simulate)
- uvm_test_2 (BFM handle based)
  - Bus Functional Model as virtual in
  - Requires dynamic vif handles
  - Result: Fail at Verilate phase
- uvm_test_cvv (industrial-style, i
  - Config class with multiple rand fie
  - Advanced UVM features (factory + config DB)
  - Result: Pass Verilate, Fail Simulate



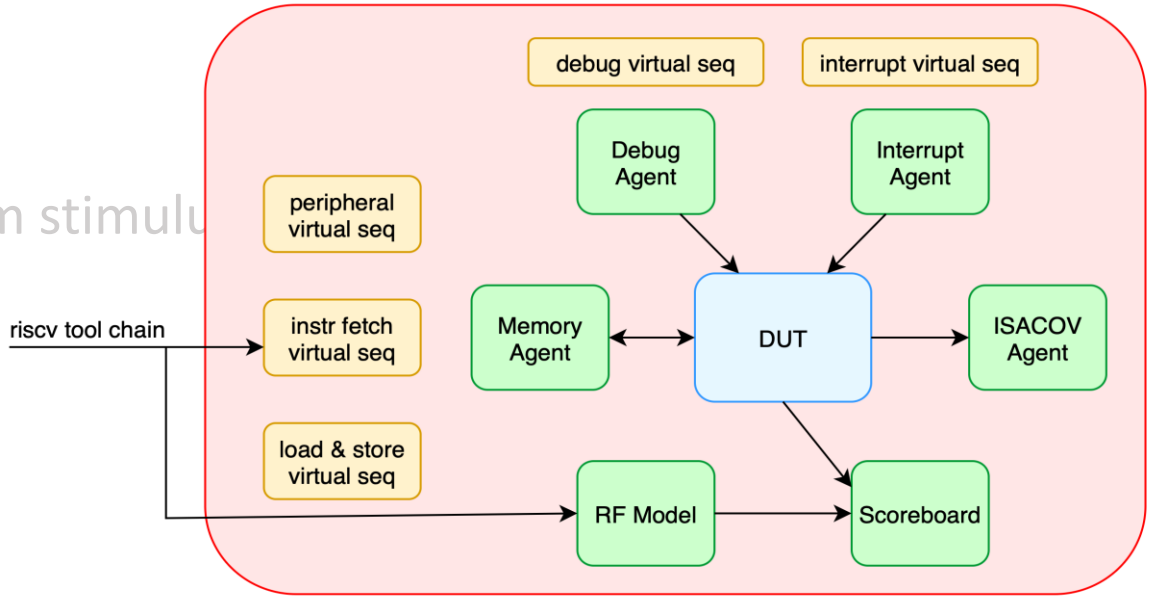
Three UVM Testbenches

- uvm_test_1 (simple agent-based)
  - Active + passive agent
  - Virtual sequence drives constrained random stin
  - Result: Pass (Verilate + Simulate)
- uvm_test_2 (BFM handle based)
  - Bus Functional Model as virtual interface
  - Requires dynamic vif handles
  - Result: Fail at Verilate phase
- uvm_test_cvv (industrial-style, inspired by OpenHW)
  - Config class with multiple rand fields
  - Advanced UVM features (factory + config DB)
  - Result: Pass Verilate, Fail Simulate



Three UVM Testbenches

- uvm_test_1 (simple agent-based)
  - Active + passive agent
  - Virtual sequence drives constrained random stimulu
  - Result: Pass (Verilate + Simulate)
- uvm_test_2 (BFM handle based)
  - Bus Functional Model as virtual interface
  - Requires dynamic vif handles
  - Result: Fail at Verilate phase



- uvm_test_cvv (industrial-style, inspired by OpenHW)
  - Config class with multiple rand fields
  - Advanced UVM features (factory + config DB)
  - Result: Pass Verilate, Fail Simulate

Results Summary

- Simple UVM environments → now supported
- More advanced UVM features → still limited
- Shows both progress and remaining gaps

Conclusion

- Extended Verilator randomization from primitives → **aggregates**
  - Arrays, structs, unions, nested combinations
  - Both **basic** and **constrained** randomization
- Optimized architecture (foreach AST, flattening, chunk encoding)
- All changes upstreamed to Verilator as open-source contribution
- Demonstrated with UVM testbenches
  - Simple → fully supported
  - Complex → partial limitations

Future Directions

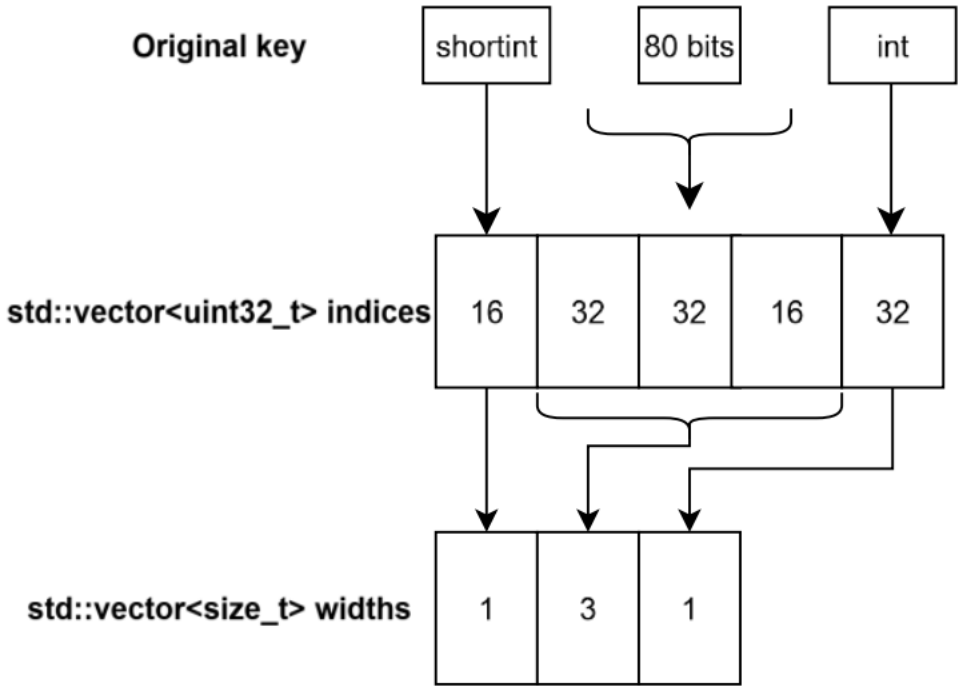
- More optimization for the current randomization process
- Still missing pieces for full UVM compatibility left to be addressed:
  - Virtual interface handling
  - Global constraints
  - Assertions
  - …
- Long-term goal → fully UVM-capable open-source simulator

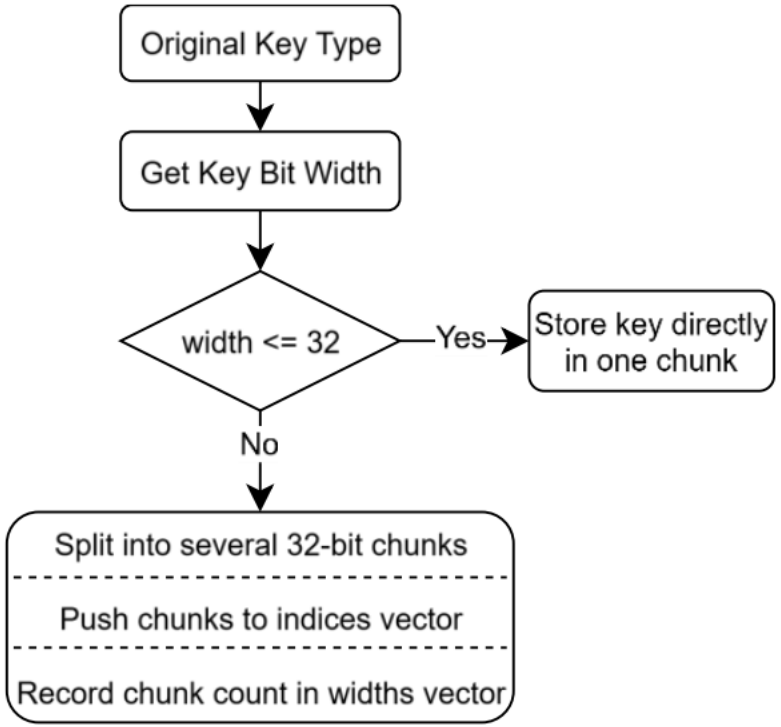
Questions

Backup --- Associative Arrays: Handling Wide Keys

- Associative arrays: index can be any data type
  - bit[1:0], int, longint, even string
- Simulator is static → must map dynamic keys into static containers
- Problem: wide keys (e.g., 10k bits or string) waste storage if container always max width
- Solution: chunk-based encoding
  - Common case (≤ 32 bits) → store directly
  - Wide keys → split into 32-bit chunks
    - e.g., 128-bit key → 4 × 32-bit chunks
  - Alongside chunks, record chunk count to reconstruct original key

Backup --- Solution: Chunk-based Encoding



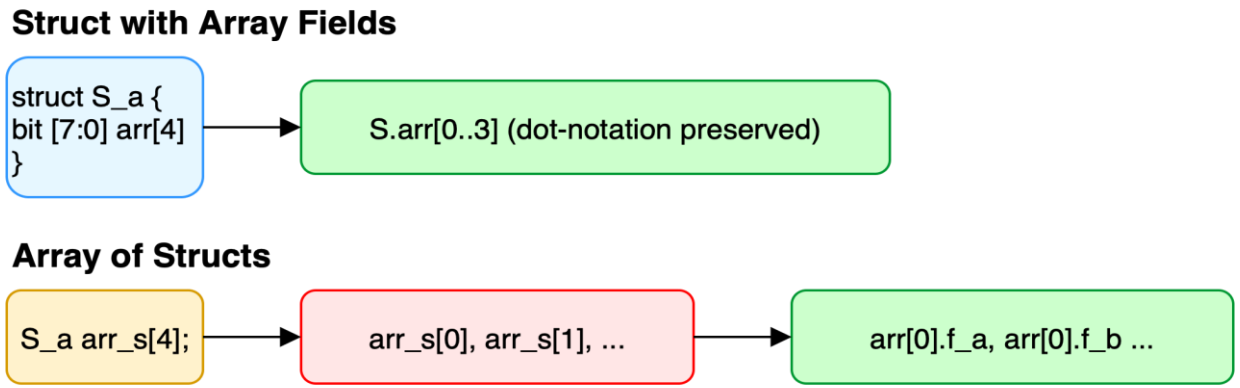


Backup --- Constrained Randomization for Structs

- Structs & Unions share one container in Verilator
- Packed structs → treated as a single long bitvector (QF_BV logic)
- Unpacked structs → need flattening into members
- SMT-LIB2: no direct support for struct types
  - Cannot declare as a whole variable
  - Must decompose into primitives
- Our solution: use dot notation (e.g., struct_a.mem1.a)
  - Allows peeling layer by layer
  - Preserves hierarchy for solver mapping and assignment

Backup --- Constrained Random for Nested Aggregated Data Types

- Struct with array fields
  - Essence: struct
  - Members may be arrays
  - Decomposed during Verilate phase
  - Keep dot-notation for array members
- Array of structs
  - Essence: array
  - First handled as array element (primitive)
  - Struct decomposition deferred to Simulate phase
- Both cases required special handling in Verilator



Backup --- Summary of Support Contributions

- Basic randomization support for All aggregated date types → recursive traversal + foreach AST node
- Constrained randomization support for All kinds of Arrays → QF_ABV + chunk-based encoding
- Constrained randomization support for Structs/Unions → flatten fields with dot notation
- Two special Nested types → Struct with array fields + Array of Structs

Backup --- The Randomization Gap in Verilator

- Commercial simulators (e.g., QuestaSim)
  - Full randomization support: primitives + aggregates(arrays, structs, unions)
- Verilator (before our work)
  - Only supports primitive types randomization
  - No support for arrays, structs, unions, nested aggregates
- Result: incomplete SystemVerilog support → UVM testbenches cannot run
- Our goal is to **support full randomization in Verilator**, moving it closer to advanced open-source verification capabilities.