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, Verification Engineer, PlanV GmbH, Munich, Germany (yilou.wang@planv.tech)

***Abstract*—As open-source verification gains momentum, Verilator has become a key tool for SystemVerilog-based simulation. However, the lack of full support for constrained randomization, a cornerstone of UVM-based verification, has remained a major limitation. This paper presents our extension to Verilator's randomization capabilities, adding support for aggregated data types such as structs, unions, and arrays, with both basic and constrained randomization. We introduce an optimized, template-based architecture that handles these complex types efficiently without compromising performance. To validate the implementation, we developed three UVM testbenches with varying structures and complexities, all verified on QuestaSim and then tested on Verilator. The results highlight both the current capabilities and remaining limitations of Verilator in supporting UVM environments. This work closes a critical gap, advancing Verilator toward a fully capable open-source solution for UVM-based verification.**

Keywords*—*Verilator; SystemVerilog; Randomization; Open-Source Verification

I. INTRODUCTION

Verilator is a widely used open-source simulator known for its high-performance, cycle-accurate simulations. It is used by many semiconductor companies for early-stage simulation, verification, and linting. However, its limited support for randomization, particularly for complex data types like structs, unions, and arrays, has been a major barrier to fully supporting SystemVerilog features and, by extension, UVM-based verification.

Randomization is essential in functional verification, enabling the generation of constrained stimulus to uncover design edge cases. Commercial simulators like QuestaSim, support full randomization for all data types, whereas Verilator only supports primitive types, which restricts its use in advanced and comprehensive verification environments.

As open-source verification gains wider traction, efforts to close this feature gap have become increasingly relevant. This paper presents our work to extend Verilator's randomization capabilities, enabling support for aggregated data types in both basic and constrained randomization cases. These improvements bring Verilator significantly closer to becoming a comprehensive open-source solution for UVM verification. All contributions have been fully upstreamed to the official Verilator repository and are publicly available.

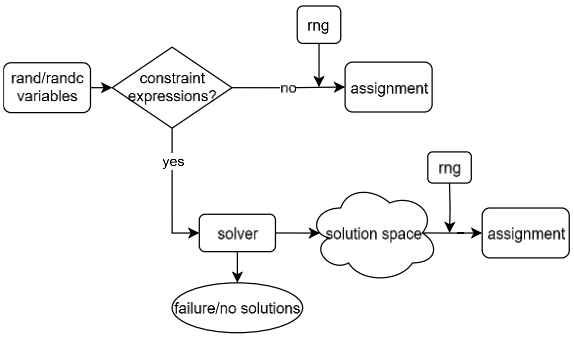
II. BACKGROUND AND RELATED WORK

A. General Randomization Process

In SystemVerilog, randomization is essential for generating diverse input stimuli during functional verification. It comes in two forms: basic and constrained. The process begins by declaring variables as `rand/randc`, making them eligible for randomization process.

In basic randomization, values are assigned directly from a random number generator (RNG), without enforcing any restrictions. Constrained randomization, on the other hand, introduces user-defined constraints that shape the range of legal values. These constraints, often expressed as inline conditions, are passed along with the variables to a constraint solver. The solver computes a solution space that satisfies all specified constraints, and the RNG picks values from this space. If no valid solution exists, the randomization fails.

Figure 1 illustrates the full process, covering both basic and constrained flows. It outlines the key steps from variable declaration to value assignment and includes pseudocode to clarify the implementation.




```
// Pseudo-code for General Randomization Process

foreach (variable in class_fields) {
    if (variable.is_rand_or_randc()) {
        mark_as_randomized(variable); } }

foreach (random_var in marked_variables) {
    if (random_var.has_constraints()) {
        solution_space = solver.solve(random_var.constraints);
        if (solution_space.exists()) {
            random_var.value = RNG.pick(solution_space);
        } else {
            report_constraint_failure();
        }
    } else
        random_var.value = RNG.generate();
}
```

Figure 1. General Randomization Process

B. *Existing Efforts for Randomization in Verilator*

Since 2020, Antmicro and other contributors have made significant progress in improving Verilator's randomization features. Key developments include:

- Internal RNG: Verilator now includes a random number generator for randomize method [1].
- CRAVE for Constrained Randomization: An external library, CRAVE was integrated to support simple constrained randomization [2].
- SMT-LIB2 Solver: The adoption of the SMT-LIB2 solver replaces CRAVE and enables more advanced constraint solving and better performance [3].

C. *Limitations*

Despite these advancements, Verilator's basic and constrained randomization remain limited to primitive types and does not support aggregated data types, like all kinds of arrays, unions, and structs.

III. EXISTING RANDOMIZATION SUPPORT IN VERILATOR

To provide context for our work on aggregated data types, we begin by summarizing the existing randomization mechanisms in Verilator. Although these features are not part of our contributions, they form the essential foundation for the extensions described in the next section.

A. *Basic Randomization for Primitive Types*

Verilator includes a built-in RNG that enables basic randomization of `rand` and `randc` variables. For primitive types such as integers and enums, the process is straightforward: the tool determines the variable's bit width, masks a long random value to match that width, and assigns the result.

B. *Constrained Randomization for Primitive Types*

Verilator supports constrained randomization through integration with SMT-LIB2-compatible solvers. During the Verilate phase [4], Verilator detects `rand` variables and associated constraint blocks in the SystemVerilog source. It then translates these constraints into SMT-LIB2 expressions and emits them via dedicated internal functions, `write_var()` for variable declaration and `hard()` for constraints. These are embedded into the generated C++ code as part of the simulation model.

In the subsequent Simulate phase, the `VlRandomizer` class calls the solver, evaluates the SMT-LIB2 expressions, and retrieves a satisfiable solution if one exists. The resulting values are then assigned back to the corresponding variables. The implementation differs between basic and constrained randomization.

Figure 2 summarizes the two flows across both Verilate and Simulate phases and includes the pseudocode to clarify the process. This mechanism currently supports primitive types and forms the foundation for more advanced randomization features in Verilator. Building upon this infrastructure, our work extends support to aggregate data types and enhance the overall randomization capability.

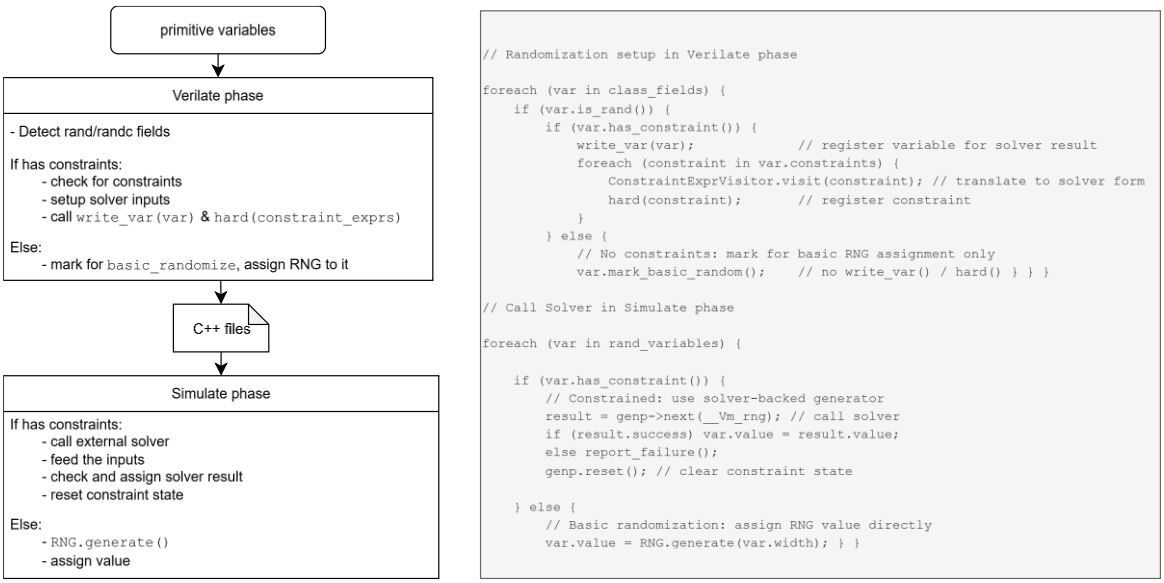


Figure 2. Implementation Flow of Basic vs. Constrained Randomization in Verilator

IV. Our Contribution: Aggregated Data Type Support

Building on the foundation described in Section III, we extended Verilator's randomization capability to support complex data structures including arrays, unions, structs, and their nested forms, to enable full randomization in Verilator. This involved both functional extensions and performance optimizations across basic and constrained cases.

A. Basic Randomization for Aggregated Types

Aggregated types such as arrays and structs can be nested to arbitrary depth and often involve multi-dimensional layouts. To support basic randomization, we implemented a recursive traversal mechanism. For each variable marked `rand`, Verilator traverses the type-hierarchy, identifying structs, arrays, or unions, and recursively processes each field until it reaches a primitive type, where RNG can be applied, as shown in Figure 3.

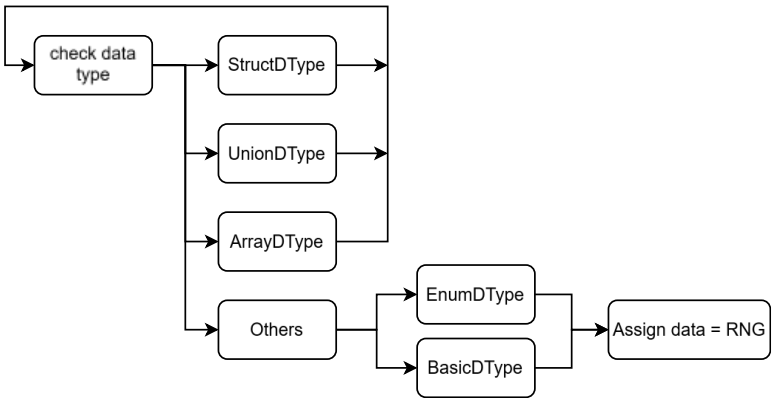


Figure 3. Flow of Basic Randomization for All Data Types

A significant challenge in this approach is controlling the size of the generated Abstract Syntax Tree (AST). Verilator compiles SystemVerilog source code in multiple stages. It starts by building an internal representation of the code, the AST, and then applies a sequence of transformation passes, such as type resolution, width inference, and randomization handling. These steps prepare the design for C++ code generation.

The problem arises because randomization is handled in an early phase of this pipeline. Fully unrolling deeply nested aggregated types at that point can lead to a substantial increase in AST size. This bloated tree consumes more memory and slows down every subsequent pass in the pipeline.

To mitigate this issue, we introduced a foreach node in the AST to abstract array iteration. Instead of explicitly unrolling every element in a multi-dimensional array up front, we insert a single foreach node that acts like a loop abstraction. It captures the traversal logic and stores metadata such as index variables and array bounds.

By doing this, we delay the actual element-wise expansion until much later in the compilation flow. This not only keeps the AST small in the early stages but also reduces redundant computation. In practice, this change significantly improves Verilator's compile-time performance for designs with large or complex data structures.

B. Constrained Randomization for Arrays

Unpacked arrays in SystemVerilog may be fixed-sized (`arr[3][4]`), dynamic (`arr[]`), queues (`arr[$]`), or associative (`arr["key"]`). These types introduce two primary challenges for constrained randomization: unknown sizes at elaboration time, and complex nested indexing with variable depths.

To address this, we implemented a recursive function `record_arr_table()` that traverses the array structure and collects metadata such as indices and bit widths. This information is flattened into a dictionary, mapping each element to a unique key. For example, a variable, declared as `rand bit [7:0] arr[2][3][4]`, produces 24 elements in total, and the element `arr[1][1][2]` is mapped to `arr_dict[18]` with a linear index in the dictionary. This allows constant-time access to any element without re-traversing the hierarchy.

Once the array is recorded, each constrained element is passed to the solver using QF_ABV logic. The full array is declared, and individual elements are accessed via nested `select` expressions. If the solver returns a satisfiable model, `store` operations are used to extract and assign the solution values. This process is repeated across all marked elements, enabling element-wise constraint solving in a structured and efficient manner. The overall simulation-time flow is illustrated in Figure 4. It shows how `VlRandomizer` performs dictionary construction, generates SMT-LIB2 expressions, interacts with the solver, and assigns the resulting values to unpacked array elements.

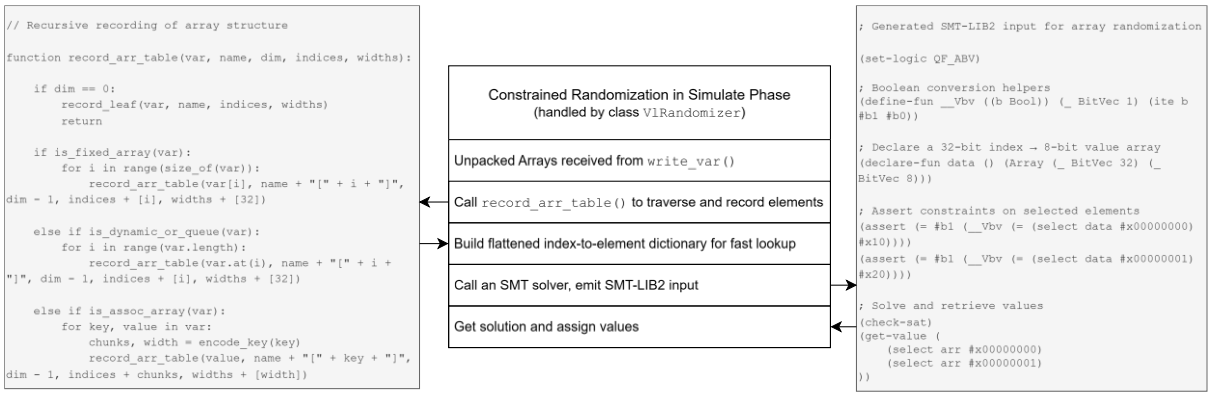


Figure 4. Constrained Randomization for Unpacked Arrays in Simulate Phase

Associative arrays present a unique challenge: their keys may have arbitrary bit widths, including types such as `bit[5:0]`, `shortint`, or even `string`. Facing this variability, one natural solution is to use larger containers to accommodate wide keys. However, uniformly enlarging the storage width would waste resources, especially since most keys in practice are still `int` or other small-width types. Therefore, we retain a 32-bit chunk format for internal storage to preserve performance for the common case and introduced a chunk-based encoding scheme, as shown in Figure 5. Keys are first examined for their bit width. If the width is less than or equal to 32 bits, such as in `shortint` or `int`, the key is stored directly in a single 32-bit chunk. For wider keys, like a custom `logic [79:0]` or long string-based indices, the key is split into multiple 32-bit chunks. Alongside the chunks, we record the number of chunks associated with each key to retain information about the original key structure. For example, an 80-bit key is encoded into three chunks: two full 32-bit values and one 16-bit remainder. This chunked encoding scheme supports efficient indexing and allows seamless translation into SMT-LIB2 expressions under QF_ABV logic, while retaining compatibility with existing array handling infrastructure.

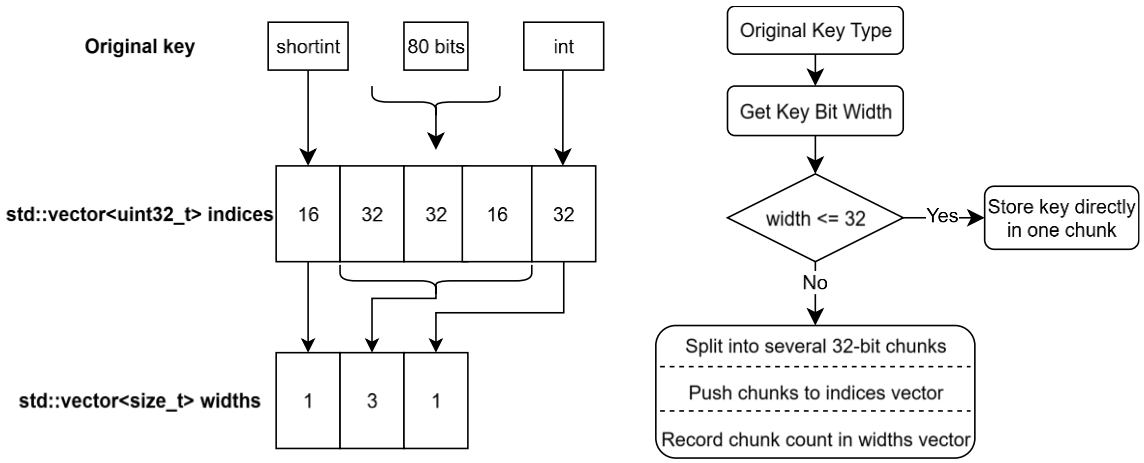


Figure 5. Chunked Encoding Scheme for Associative array's Index

C. *Constrained Randomization for Structs*

Structs in SystemVerilog fall into two categories: packed and unpacked. Packed structs are treated as contiguous bitvectors, allowing them to be randomized like primitive types using QF_BV logic in SMT-LIB2. Verilator can directly generate constraints for these variables and solve them as a whole.

Unpacked structs, however, are more complex. Unlike arrays, which benefit from QF_ABV logic for element-wise access, SMT-LIB2 offers no dedicated bitvector-level logic extension for hierarchical named fields in structs. As such, there is no straightforward way to "extend" the existing logic to support structs in the same manner. This absence has been a key obstacle to implementing constrained randomization for structs in Verilator.

Despite this difference, the underlying idea is the same: both arrays and structs are aggregates. For arrays, our approach is to decompose them into elements, track their dimensions and indices, and pass each element individually to the solver. Applying the same logic to structs, we decompose them into individual members (fields). The only difference is that instead of using index-based access as in arrays, struct fields are accessed using the dot (.) operator, which allows us to preserve hierarchical naming. Table I summarizes how we handle different aggregate types in relation to SMT-LIB2 capabilities in Verilator:

Table I. Comparison of aggregate types in SystemVerilog and their SMT-LIB2 handling

Category	***Packed Struct / Packed Array***	***Unpacked Array***	***Unpacked Struct***
Behavior	Continuous bitvector	Indexed collection with dimensions	Loosely grouped named members (fields)
SMT-LIB2 Support	QF_BV	QF_BV + array logic (QF_ABV)	No dedicated bitvector-level logic extension
Randomization	Treated as a single variable	Element-level access via select/store	Needs member-level flattening into separate variables
Our Solution	Direct pass to solver	Flatten, track indices, element-wise solve	Decompose via dot (.) operator-based naming, register each member

To support unpacked structs in this context, we introduced a flattening strategy during the Verilate phase. Structs are recursively traversed, and each randomizable member is extracted and assigned a dot-connected hierarchical name (e.g., `struct_a.mem1.a`). These flattened names are used to individually emit SMT-LIB2 variable declarations and constraints.

An important distinction with structs, unlike other data types mentioned earlier, is that the IEEE 1800-2023 standard [5] allows two styles of randomization:

- If the whole struct is marked `rand`, all its members are randomized.
- If only some members are individually marked `rand`, only those are considered in randomization process.

This flexibility is essential for real-world verification environments. As shown in the left part of Figure 6, struct `data_s` is flattened into `data_s.a` and `data_s.b`, while the member `c` is ignored because it is not marked with the `rand` qualifier. Originally, Verilator had only coarse-grained detection for `rand` annotations. To support unpacked structs with finer granularity, we introduced two helper member functions:

- `markConstrainedRand()` – to collect all fields that should participate in constraint solving.
- `isConstrainedRand()` – to query whether a particular field should be randomized.

Together, these functions enable a structured and hierarchical decomposition of unpacked structs. The pseudocode summarizes this process during the Verilate phase, as shown in the right part of Figure 6. Each marked field is individually registered using `write_var()`, preserving its hierarchy via dot-connected naming. Constraints are translated to SMT-LIB2 and passed via `hard()`.

This flattening strategy treats each struct field as an independent variable while maintaining its structural context. As a result, Verilator seamlessly supports unpacked structs in a solver-compatible way.


```
// Marking and Decomposing in Verilate Phase

// Detect and Mark
if field is marked 'rand':
    if field is a struct:
        // Case 1: full struct marked rand, mark all its members
        for each member in field:
            markConstrainedRand(member)
    else if fromp is a struct:
        // Case 2: struct member individually marked rand
        markConstrainedRand(field)

// Decompose: resolve dot (.) operator-based naming
if isConstrainedRand(field):
    dot_name = fromp_name + '.' + field_name
    con_expr = convert_to_smtlib2_format(constraint_expr)
    // Emit variable and constraint
    write_var(dot_name)
    hard(con_expr)
```

Figure 6. Handling Structs in Verilator's Constrained Randomization Flow

D. Constrained Randomization for Nested Aggregated Data Types

After establishing support for structs and arrays, we extended our framework to handle nested combinations of these types. Two common patterns were addressed, struct with array fields and array of structs.

For cases where a struct contains array fields, our existing logic remains largely applicable. During the Verilate phase, the struct is decomposed using dot-connected naming, and the array member is passed as a complete unit. Relevant metadata, such as dimensions and widths, is extracted during this step to ensure correct downstream handling. In the Simulate phase, the array field is randomized using the same mechanisms developed for standalone arrays. This approach allows seamless integration of array types within struct definitions without introducing additional complexity.

More complexity arises when arrays contain structs as elements. Since struct decomposition occurs during Verilate phase, but arrays are processed during Simulate phase, a timing mismatch exists: the struct cannot be flattened before the array is known. To resolve this, we shifted the decomposition of Array's struct elements into the Simulate phase. During runtime, we detect whether an array element is a struct, and if so, dynamically traverse and register its fields. Each field is assigned a dot-connected name like `arr[1][3].a` and `arr[1][3].b`. These fields are then individually registered and randomized.

To support this mechanism, we implemented a new recursive utility that mirrors our earlier `record_arr_table()` function. This new handler, `record_struct_arr()`, ensures proper traversal and registration of all struct members, even in multi-dimensional or mixed-type array contexts. As a result, Verilator

now supports constrained randomization for deeply nested aggregated data types, covering both directions of nesting.

V. Results: Three UVM Testbenches Run in Verilator

With the support for full randomization in Verilator, advanced open-source verification has taken another solid step forward. To demonstrate the practical impact of our work, in this section we evaluate Verilator's capability to support UVM-based verification using three representative UVM testbenches with varying structures and complexity.

To better illustrate the evaluation setup, we briefly describe the structure and intent of each testbench. Although each testbench follows standard UVM methodology, they were deliberately chosen to reflect different levels of integration complexity and randomization usage.

`uvm_test_1`, inspired by Qiang Zhang's *UVM Combat* [6], adopts a basic layered architecture. It includes one active agent (with sequencer, driver, and monitor) and one passive agent (monitor only), interfacing with the DUT through two virtual interfaces (`vif_in`, `vif_out`). A virtual sequence inside the uvm_test component drives the constrained random stimulus via `uvm_do()`. This test verifies the full flow of constrained randomization and successfully passes both the Verilate and Simulate phases.

`uvm_test_2`, following the structure from Ray Salemi's *The UVM Primer* [7], introduces a bus functional model (BFM), implemented as a virtual interface that serves as a software–hardware bridge. Both the driver and monitor call BFM functions to interact with the DUT. While this is a common pattern in SystemVerilog UVM, Verilator currently lacks full support for dynamically generated virtual interface handles, causing the test to fail during Verilate phase.

`uvm_test_cvv`, loosely modeled after the OpenHW Group's core-v-verif repository [8], resembles a more advanced industrial-style testbench. A key feature is the use of a configuration class (`cfg`), implemented as a `uvm_object` with multiple `rand` fields controlling agent behavior, coverage collection, logging, and more. These objects are randomized at the top level, typically in the `uvm_test`, and propagated through the UVM factory and configuration database down to the env, agents, and subcomponents. This modular, reusable configuration pattern is widely adopted in real-world UVM environments for flexibility and scalability. While the testbench passes Verilate phase, it fails during Simulate phase due to Verilator's incomplete support for global randomization state, inherited class constraints, and context-sensitive configuration flows. These limitations surfaced when attempting to randomize class objects at the test level and apply the results consistently across multiple components. Such flows are essential for scalable, reusable testbenches and represent an important direction for future Verilator development.

Table II summarizes the results and highlights the current boundaries of Verilator's UVM support. All three testbenches were first validated in QuestaSim to ensure full compliance with SystemVerilog and UVM standards. The complete testbenches and supporting files have been made publicly available [9] to encourage reproducibility and further exploration by the community.

Table II. Running UVM Testbenches with Verilator

Testbench	Verilate Phase	Simulate Phase	Remarks
uvm_test_1	Pass	Pass	Minimal structure, fully supported
uvm_test_2	Fail	N/A	Uses BFM handles, unsupported
uvm_test_cvv	Pass	Fail	Advanced UVM features, runtime issues

VI. Conclusion and Future Directions

This work extends Verilator to support full randomization, including both basic and constrained, for aggregated data types. A templated framework ensures high performance while handling complex structures, and all changes

have been upstreamed via multiple pull requests. The complete implementation, along with all related tests and examples, is publicly available as part of the open-source Verilator repository [10] and the PlanV verification suite [9].

Experiments with three UVM testbenches demonstrate that Verilator can now successfully execute simple UVM environments, particularly those relying on randomization. Key UVM constructs, such as `uvm_do()`, are now fully functional with complex randomized types, including structs, arrays, and their nested forms.

By enabling full randomization and extending support beyond primitive types to arrays and structs, this work fills a critical gap in Verilator and significantly advances the capabilities of open-source verification. However, full UVM support remains a longer-term goal. Results with three UVM testbenches also show that many important features, such as virtual interface handling, global randomization state, and class-based constraint inheritance, are still limited or incomplete, and remain open for future improvement.

Our contribution lays the groundwork by expanding the scope of data types that can participate in randomization, but achieving full UVM compatibility will require continued contributions from the open-source community and sustained maintenance. Reaching that milestone would represent a major leap forward in enabling advanced open-source functional verification workflows.

REFERENCES

[1] Verilator. (2020). randomize() class method. GitHub. https://github.com/verilator/verilator/pull/2607.

[2] Antmicro. (2024). Introducing constrained randomization in Verilator. Antmicro Website. https://antmicro.com/blog/2024/03/introducing-constrained-randomization-in-verilator/.

[3] Verilator. (2023). Constrained randomization with popen and external solvers. GitHub. https://github.com/verilator/verilator/pull/4947.

[4] PlanV. (2024). Enabling UVM Support in Verilator Series — Basic Randomization Support for Aggregate data types, PlanV Website. https://planv.tech/2024/11/07/enabling-uvm-support-in-verilator-series-basic-randomization-support-for-aggregate-data-types/.

[5] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017) , vol., no., pp.1-1354, 28 Feb. 2024, doi: 10.1109/IEEESTD.2024.10458102.

[6] Qiang Zhang, UVM Combat, Machinery Industry Press, 2014.

[7] R. Salemi, The Uvm Primer: A Step-By-Step Introduction to the Universal Verification Methodology, Boston Light Press, 2013.

[8] OpenHW Group. (2024). OpenHW CORE-V Verification Strategy, OpenHW Group Website. https://docs.openhwgroup.org/projects/core-v-verif/en/latest/index.html#openhw-group-core-v-verification-strategy.

[9] PlanV. PlanV_Verilator_Feature_Tests, GitHub Repository. https://github.com/planvtech/PlanV_Verilator_Feature_Test, accessed June 2025.

[10] Verilator. Verilator master branch, GitHub Repository, https://github.com/verilator/verilator , accessed June 2025.