Enhanced Verbosity Methodology

Gergő Vékony, József Mózer
ARM, Budapest, Hungary
gergo.vekony@arm.com, jozsef.mozer@arm.com

***Abstract*—This paper introduces an Enhanced Verbosity Methodology (EVM) for Universal Verification Methodology (UVM) environments to address the usability and scalability challenges of controlling report verbosity in large, vertically integrated testbenches. EVM enables dynamic, runtime updates of message verbosity to reduce log noise and improve simulation performance by filtering messages before they are constructed. The approach supports gradual rollout and backward compatibility via an opaque class-extension technique that injects an EVM layer into class inheritance chains without disrupting existing UVM hierarchies.**

EVM organizes messages into purpose-driven categories (build, header, detail, item, and report), stores per-component verbosity in local configuration objects, and maintains centralized registries keyed by component group, type, instance name, and full hierarchical name. Verbosity settings are supplied in an external JSON descriptor parsed via DPI-C, with hashed identifiers used for efficient matching and consistent simulator performance. A case study reports rapid adoption in a subsystem-sized environment, cross-simulator viability, and improved debug turnaround through triggered verbosity escalation during simulation.

***Index Terms*—Design Verification, Dynamic Verbosity Control, Gradual Rollout, IEEE-1800.2, UVM, User Experience, Vertical Integration**

I. Introduction

In the EDA industry, new tools are released multiple times every year, providing enhanced debug and verification capabilities driven by the ever more complex IP and system development. New paradigms emerge around design and testbench partitioning, coding guides describe performance-oriented solutions, and AI-assisted tools increasingly handle routine tasks. One area where progress is noticeably slower, however, is the interface and communication between the software and its users. Whether we consider the user manuals, logging and feedback options, or graphical interfaces, it is evident that the User Experience (UX) is often not the highest priority.

This problem is especially acute in verification, where textual output describing stimulus and observed results is typically stored in log files and searched. UVM [1] provides several ways to control what is displayed: global verbosity threshold, per-component verbosity settings, and report server customization. Many approaches exist for controlling these settings (including simulator-specific features from vendors); however, a solution that is both user-friendly and suited for vertical reuse has not been widely adopted.

While this may not be a major issue in a single (often unit-level) testbench, it becomes a significant challenge at higher integration levels, where a system or subsystem testbench can contain tens or hundreds of predictors, checkers, and agents. In such environments, it can take substantial effort to configure components to deliver sufficient information without crippling performance or flooding logs with noise. Ideally, every unit-level verification environment would provide a mechanism to suppress its own messages, but more often, this is not implemented due to complexity or time constraints.

This is especially true for inherited or third-party components, where the owners may have had different visions, used their own filtering mechanisms, or relied on a specific UVM report server implementation, or even had an external framework that handles these issues. Nevertheless, practical needs remain to validate hypotheses, to demonstrate that block-level errors are not observable on the integration level, or to trace back integration-level failures to block-level side effects; a solution is still very much required.

In this paper, we propose an Enhanced Verbosity Methodology (EVM) that provides:

1) A reusable methodology without proprietary components, suitable for any UVM-based environment,
2) A method to dynamically control and update UVM message verbosities,
3) A method to store and distribute verbosity settings across different vertical integration levels,
4) A way to separate and categorize UVM messages based on their purpose,
5) An implementation orthogonal to UVM report server and report catcher, and
6) A method that can be rolled out quickly and flexibly, while preserving backward compatibility.

In EVM, we employ a collection of UVM-extended components with type parametrization and a lightweight (handle-based) component registry. We define a JSON descriptor format that can specify verbosity settings for components by group, class type, instance name, or full hierarchical name (FHN). Because the descriptor is not

compiled but parsed/processed at runtime, users can rerun simulations and refine settings quickly. We also provide helper macros that retrieve category-specific verbosity values from each component's local configuration object, enabling dynamic verbosity control. To reduce lookup and comparison costs, we use string hashing for matching. The following sections describe these features in more detail.

II. EVM Paradigms

As described in the introduction, the motivation behind EVM was to provide a way to dynamically control the verbosity values of UVM-based verification components. To achieve this, we defined the following set of paradigms that guided us during the design and implementation of EVM.

A. Dynamic Rollout

One of the greatest issues in any project is to introduce a set of fundamental changes that affect its codebase or hierarchy in one way or another. Sometimes it is not feasible to do so due to its unseen side effects or the number of changes it requires, sometimes it is even impossible as some code might be required to be kept in a frozen state. SystemVerilog also does not allow multiple inheritance, so adding functionality between classes in inheritance chains might not be trivial.

Fortunately, this can be overcome by the *opaque class extension method* described in [2], and it solves several obstacles for EVM. It allows us to inject a class into the inheritance chain between our targeted class and its parent, thus making the EVM functionality available for our target and all of its descendants. It also relies on the constructor limitations of SystemVerilog (in [3], Ch.8.7 *Constructors* and Ch.8.15 *Super*) as the constructor of the parent class is guaranteed to be executed before the descendants. This gives us the opportunity to set up EVM for the targeted class before any of its code is executed, including all UVM phases.

Our implementation also does not require creating and maintaining a full layer of classes to enroll different components in EVM, the opaque method makes it possible to inject functionality into any UVM component from a single class definition. As EVM adds a new layer of options over UVM, it is transparent and preserves backward compatibility. Thus EVM can be rolled out for components gradually, and even in those, verbosity categories can be implemented progressively.

For data separation reasons, EVM instantiates a configuration object for all EVM-aware classes to encapsulate all the settings, flags, and verbosity values locally without polluting the contents of its descendants and introducing the need to look up values from a centralized registry or database. In the case where no verbosity descriptor is provided, EVM can use the EVM-package defined default values, or the group defaults can be set up directly from the testbench via EVM public API calls.

B. Convenience

No matter how good a methodology is from a technical perspective, it will fail if the application and its limitations are not precisely defined or if it is too convoluted for deployment or daily usage. For this reason, we have chosen JSON for the default verbosity descriptor format of EVM, as it is text-based, version-control compatible, and easy to read, property order-insensitive, and can easily be extended in the future. We debated implementing an additional command line processor for verbosity descriptors, but as engineers tend to store those (usually) complex arguments in other note-taking applications, it was therefore chosen to guide them to a file-based method.

Message verbosities can be set using four levels of granularity, ranging from the broad category of group, class type, and instance name to specific full hierarchical name. As verbosity updates will follow this granularity order, it is straightforward to come up with rules to, for example, suppress all messages from the group members of `EVM_AGENTS`, while keeping a specific agent or several agents instantiated from the same class above the verbosity limit.

Verbosity descriptors can be moved and reused between different vertical levels because, except for the FHN entries, they can be reused directly. Nevertheless, the same JSON can contain FHN settings both for unit- and integration-level testbenches, as FHNs not found in the current environment will be simply ignored.

For the sake of simplicity, the JSON parsing and string hashing are done through the UVM DPI-C calls by open-source libraries, so no SystemVerilog implementation was required. While the C code compilation might be required to be added to the compilation flow, as stable libraries are used, they could also be distributed as precompiled objects.

III. Other Considerations

A. Filtering of Messages

When a subsystem- or system-level testbench is built from block-level environments, usually there are two problematic areas: the verbosity of report messages and the grading of coverage, as both are done from the perspective of their respective origin. Something falling under the category of `UVM_LOW` or `UVM_HIGH` might not be true on a higher integration level, and it is hard to differentiate between what is useful and what seems useless. There are ways to filter out messages that only add noise, for example, using the `uvm_report_catcher` and various callbacks. For this, some kind of message uniformity might be required to match against in the `ID` or `MSG` fields of the callbacks. If such solutions were not implemented, the component might be suppressed altogether.

Filtering also presents a performance problem, as the message filtering is done after the messages are created and before they are printed. On a higher vertical level, not only the number of RTL and verification components increases, but also their costly operations – like string substitutions, I/O activities, etc. – and this degrades simulation performance significantly. Thus, it is imperative to try to come up with a solution that can filter messages before they are created to keep computational resource use close to a minimum. This was the motivation to implement EVM as a dynamic verbosity modification method, where the verbosity check happens before the message creation.

Also, a solution that is orthogonal to the item/object based verbosity escalation method described in [4] and already used at ARM seemed like the proper choice.

B. Message Categories

UVM, as a framework, does not specify how to use the reporting macros at all. It is up to the engineers to come up with rules or a guide on how to mark important messages and how to separate them from the rarely useful ones. Methods like differentiation done by the message `ID` or `VERBOSITY` field might be common or maybe a configuration variable is implemented to guard the `sprint()` of the `uvm_sequence_item` derived classes. No matter what method is used, it seems required to have different layers of importance set in any components.

In EVM, the message categories were introduced to give a vague, high-level classification for the users to mark the intention of a given message. In different parts of the verification environment's lifecycle or usage, different messages seem to be valuable; like the feedback provided by the build and connect phases quickly devalues after the initial bringup and instantiation pattern set-ups are done. Similarly, in some cases, it might be valuable to see the function call order of a given predictor but not the actual prediction messages or exact values.

For this reason, we introduced the set of `build`, `header`, `detail`, `item`, and `report` categories and a separate verbosity level for them. With this implementation, it is possible to mute a given category completely, while leaving the rest of the messages intact for the report catcher's callbacks to do the final filtering. With five categories, it is possible to separate the messages of the build and cleanup phases from the run phases and give the option to further break down the run phases messages based on their importance and length. Further categories can be added in the future, if required, as category verbosity values are picked up from the local configuration object with a simple macro.

C. Local Configuration Objects

As EVM is required to keep track of the EVM-aware components to be updated when a verbosity descriptor is processed, we employed a centralized registry of *handles* pointed to the EVM configuration objects. With this implementation, we exchanged the performance costs of verbosity value lookups from a centralized database – which happens many times during simulation – to the memory cost of the configuration object and a handle, while keeping the possibility to iterate over the registry contents from the registry itself. This implementation guarantees that both the updates and lookups are done quickly and efficiently.

To further cut down computational costs, we not only implemented one, but four registries: based on the component's group, class type, instance name, and full hierarchical name. It provided benefits during the implementation and bringup of EVM; and also it reduces the number of registry iterations required during verbosity update calls.

D. String Hashing

The use of string hashes instead of strings was an EVM implementation choice, based on the results of testing done with the three biggest vendor simulators. In some cases, a certain simulator decided to use strings as strings instead of applying some internal hashing onto them. Furthermore, in several occurrences only the first few characters of the string were used to generate the associative array key hash, resulting in numerous collisions due to the nature of hierarchical path names. Experiments showed that this decision is based on the length of the longest string of an array and it has a severe impact on lookup and comparison performance that is measurable in seconds of runtime. While this seemed like a minor annoyance, we decided to enforce string hashing in all cases. This resulted in very consistent runtime metrics, independent of the used string lengths.

IV. EVM Rollout and Implementation

A. EVM Layer Rollout

From the perspective of the component being made EVM-aware, the following code changes are required in the target class (highlighted in bold).

1) The class extension syntax must be changed to incorporate the **evm_layer** inheritance:

```
//  From: class amba_axi_env extends uvm_env;

class amba_axi_env extends evm_layer #( uvm_env );
```

2) The **evm_group** must be set up in the constructor (if it is to be used), as it is required by the component registration happening in the **build_phase**():

```
function amba_axi_env::new(string name = "amba_axi_env", uvm_component parent);
  super.new(name, parent);

  set_evm_group(evm_types::EVM_ENV);

  //  ...existing code...
endfunction: new
```

3) And finally, it must be ensured that the component calls **super.build_phase**() to allow the EVM registration to happen:

```
function void amba_axi_env::build_phase(uvm_phase phase);
  super.build_phase(phase);

  //  ...existing code...
endfunction: build_phase
```

These requirements help to keep the number of invasive code changes to a minimum and do not require touching the parent of the extended component at all.

B. EVM Layer Implementation

The **evm_layer** is the opaque class that adds the EVM-related functionality; the configuration object and the registration handling for the EVM-aware components. UVM factory registration was omitted from this class, as virtual class factory registration is not available on older versions of UVM.

The class is defined as follows:

```
virtual class evm_layer #(type WRAPPED_T = uvm_component) extends WRAPPED_T;
  protected evm_comp_config evm_config_obj;
  protected evm_types::evm_comp_group_t evm_group;
```

It is worth noting that the **evm_layer** injection does not change the behavior of any already implemented **type_override** and **inst_override** methods, as **evm_layer** becomes the *parent* of the extended class and thus preserves backward compatibility between *descendant* classes and the extended one.

```
function evm_layer::new(string name = "evm_layer", uvm_component parent);
  super.new(name, parent);
  this.evm_config_obj = get_evm_config_obj();
endfunction: new
```

SystemVerilog LRM [3] in Chapter 8.15 states: *"A **super.new** call shall be the first statement executed in the constructor."* This behavior can be exploited to ensure that the EVM layer's constructor is processed before the constructor of its descendants, first of all, the EVM extended class.

However, this also presents a limitation, as the EVM-aware component cannot yet be pushed to the registry as the **evm_group** value was not yet set. (If it was not set as an initial value.) As the coding guides of ARM discourage the use of initial values, we circumvented this issue with the addition of the **evm_group** variable and a **build_phase**() time registration.

The EVM configuration object is declared as **protected**, and the instantiation is embedded into its **getter**() function, thus incorporating multiple EVM layers into the class hierarchy will not cause ambiguous variable resolutions. It also stores the full hierarchical name and **inst_id** of its descendant for debug purposes.

```
function evm_comp_config evm_layer::get_evm_config_obj();
  if (this.evm_config_obj == null) begin
    evm_config_obj = new(this.get_full_name(), this.get_inst_id());
  end
  return this.evm_config_obj;
endfunction: get_evm_config_obj
```

The **build_phase**() of the EVM layer takes care of calculating the hashes related to the component and the registration process. By that time the descendant constructors have been finished, the **evm_group** value is set up and ready to be used.

```
function void evm_layer::build_phase(uvm_phase phase);
  super.build_phase(phase);

  this.evm_config_obj.set_evm_group(evm_group);
  this.evm_config_obj.set_evm_hash_type(evm_hasher::generate_hash(...));
  this.evm_config_obj.set_evm_hash_name(evm_hasher::generate_hash(...));
  this.evm_config_obj.set_evm_hash_path(evm_hasher::generate_hash(...));

  evm_layer_globals::register_component(this.evm_config_obj);
endfunction: build_phase
```

And with that, the component is made EVM-aware and ready to be used.

C. EVM Layer Globals Implementation

The **evm_layer_globals** *virtual class* is the core of EVM. It provides the public API of EVM, stores the handles to the EVM-aware component configuration objects, handles the DPI-C based JSON descriptor processing calls and updates, and provides debug functionality. It is declared as:

```
virtual class evm_layer_globals;
  //  public API
  extern static function longint unsigned get_evm_inst_count();
  extern static function void init_group_defaults(
    input evm_types::evm_comp_group_t group,
    input evm_types::evm_verb_struct_t group_verbosities
  );
  extern static function void register_component(ref evm_comp_config comp_config_obj);
  extern static function void load_verbosity_descriptor(input string fname);
  extern static function void update(evm_types::evm_descriptor_t descriptors[]);
  extern static function void print_topology();
```

Most of these functions are self-explanatory, but there are several things to be noted:

1) The **init_group_defaults**() makes it possible to override the hardcoded default of verbosities of **evm_pkg** even before any verification components are instantiated or UVM's **run_test**() is called. This is due to the reason that the **evm_comp_group_t** is defined as an enum and stored in a static array.

2) The **register_component**() takes the EVM configuration object, pushes it into the registry queues, and applies the group verbosity default values. As registration takes place during the beginning of a component's **build_phase**(), to suppress or change the **build** category messages of that phase, an early descriptor processing (**load_verbosity_descriptor**()) or a group verbosity initialization (**init_group_defaults**()) might be required.

3) As there are no limitations enforced on how many and what kind of **load_verbosity_descriptor**() and **update**() calls are allowed, this does not imply any restrictions for the previously mentioned case.

The rest of the variables and internal functions were made private as they are not expected to be accessed or modified by the testbench.

The EVM registry contains arrays of queues. The group array is a fixed-length one, while the rest are associative ones, indexed by the respective hash values of used underlying strings. The queue dimension contains the handles to the EVM configuration objects.

```
static protected evm_comp_config evm_group_registry[evm_types::evm_comp_group_t][$];
static protected evm_comp_config evm_type_registry[evm_types::evm_hash_t][$];
static protected evm_comp_config evm_name_registry[evm_types::evm_hash_t][$];
static protected evm_comp_config evm_path_registry[evm_types::evm_hash_t][$];
```

During the **update**() call, the verbosity descriptor is processed by **read_descriptor_file**() and a subsequent DPI-C call to the C-based **yyjson** [5] parser. The results are pushed to the four update queues based on the **scope_type** values of the JSON descriptor elements.

```
static protected evm_types::evm_verb_struct_t evm_updates_by_group[evm_types::evm_comp_group_t];
static protected evm_types::evm_verb_struct_t evm_updates_by_type[evm_types::evm_hash_t];
static protected evm_types::evm_verb_struct_t evm_updates_by_name[evm_types::evm_hash_t];
static protected evm_types::evm_verb_struct_t evm_updates_by_path[evm_types::evm_hash_t];
```

After the update queues are populated, a simple iteration is used to update all matching registered configuration objects hashes with the new verbosity values. As component hashes are calculated during component registration and also during the verbosity descriptor processing, no complex comparisons are needed here.

```
foreach (registry[reg_hash]) begin
  foreach (updates[upd_hash]) begin
```

```
    //  If the hashes do not match; continue
    if (reg_hash != upd_hash) continue;
    //  Otherwise apply the updates for the matching components
    comp_q = registry[reg_hash];
    foreach (comp_q[i]) comp_q[i].set_verbosities(updates[upd_hash]);
  end
end
```

D. EVM Component Configuration Object

The **evm_comp_config** is the third major class of EVM, it contains a local set of verbosity values to be picked up by its owner, the hashes of its owner's identification strings and debug variables. The class is not a descendant of **uvm_object** as it serves as a simple container and most likely not participating in type overrides.

```
class evm_comp_config;
  protected string evm_owner_full_name;
  protected int unsigned evm_owner_inst_id;

  protected evm_types::evm_comp_group_t evm_group;
  protected evm_types::evm_hash_t evm_hash_type;
  protected evm_types::evm_hash_t evm_hash_name;
  protected evm_types::evm_hash_t evm_hash_path;

  protected int unsigned evm_verbosity_build;
  protected int unsigned evm_verbosity_header;
  protected int unsigned evm_verbosity_detail;
  protected int unsigned evm_verbosity_item;
  protected int unsigned evm_verbosity_report;

  extern function void set_evm_group(evm_types::evm_comp_group_t evm_group);
  extern function void set_verbosities(evm_types::evm_verb_struct_t verb_struct);
endclass
```

There are several things to be noted here.

1) All of the **protected** members are accessible with getter and setter functions. They should either be handled in the **build_phase()** and during the EVM component registration (e.g., group and hashes); by the **evm_layer_globals** functions (e.g., **set_verbosities()** during verbosity updates) or should be indirectly accessed by helper macros, as in the case of the individual message category verbosities.

2) The **set_evm_group()** is a special setter function, not implemented with a simple getter-setter macro for protected members. The reason behind this choice was to provide a feedback option for the user, if **set_evm_group()** was called after the EVM registration, as changing the group type will have no effect on the component behavior. (Group verbosities will be picked up from which group queue the configuration object has been registered to, not based on this variable.) This might seem to be an inflexible implementation, however changing the **evm_group** value (later) to allow suppression or special handling of a given class instance is equivalent of giving the class or instance an entry in the JSON verbosity descriptor.

3) The **set_verbosities()** processes a verbosity structure (**evm_verb_struct_t**) received from a parsed JSON descriptor and extracts the **struct** fields into the separate verbosity variables, like **evm_verbosity_build** etc. This choice simplifies GUI-based debugging, as it avoids expanding the structure to inspect individual verbosity fields.

E. EVM Macros

The final auxiliary component of the EVM package are the macros to pick up the message category verbosity values, stored in the local component configuration objects, for example:

```
`define EVM_BUILD this.evm_config_obj.get_evm_verbosity_build()
```

As this macro will always pick up the actual verbosity value from the component's EVM configuration object, it ensures that any verbosity updates will be reflected in the UVM message macros and therefore achieves the dynamic verbosity control goal of EVM.

```
`uvm_info(get_name(), "build_phase(): Completed successfully!", `EVM_BUILD)

`uvm_info(get_name(), $sformatf(
  "arch_predict(): AXI transaction with inst_id: 0x%0h received for processing.",
  item.get_inst_id()
), `EVM_HEADER)

`uvm_info(get_name(), $sformatf(
  "axi_coerce(): AxCache of item with inst_id: 0x%0h has been coerced from %s (0x%0h) to %s (0x%0h).",
  item.get_inst_id(), item.get_axcache().name(), item.get_axcache(), coerced_axcache.name(), coerced_axcache
```

```
), `EVM_DETAIL)
```

F. DPI Layer

The DPI layer of EVM consists of two distinct parts, one that serves as an interface for the descriptor parser and one that provides access to the hasher algorithm.

We have chosen a well-known and tested open-source JSON parser, **yyjson** [5] that is written in ANSI-C (C89), and has proper implementations for both **x86-64** and **AArch64** architectures to remove possible cross-platform and compiler compatibility issues. For the hashing algorithm, interfaces have been provided for both the 32-bit and 64-bit versions of **xxHash** [6] and the GNU Coreutils [7] **md5sum**. Based on testing and profiling results, we concluded that using hashes might have an advantage in certain cases with certain simulators versus using raw strings (see Section III/D), but the actual performance difference of the hash algorithms is insignificant from the simulation perspective. All modern hashers can provide negligible collision risk for our use case.

G. JSON Descriptor Format

The **yyjson** library does not enforce any specific JSON format, it only provides parsing capabilities. For this reason, we defined a simple JSON format for the verbosity descriptors.

An example verbosity descriptor pseudo code is shown below:

```
<!-- EVM verbosity descriptor format -->
[
  {
    <!-- scope_type enum is {group, name, full_name, type} -->
    "scope_type": 1,
    "scope": "amba_axi_env",
    <!-- verbosities are [build, header, detail, item, report] -->
    <!-- Note: suppressing everything but the header type here -->
    <!-- Note: 100 corresponds to UVM_LOW, 600 is above UVM_DEBUG -->
    "verbosities": [600, 100, 600, 600, 600]
  }
]
```

The descriptor is parsed by **yyjson** and the scope string is processed by the same hasher algorithm as used on the SystemVerilog side. The resulting struct is passed back to SystemVerilog via DPI-C and pushed into the **evm_layer_globals** update queues based on its **scope_type**.

Further improvements could be made to modify the format to allow using enum strings instead of integers for the **scope_type** and **verbosities** values.

V. EVM Integration Case Study

A. Integration

After the initial implementation of EVM, we measured how much effort it took to enroll an ARM subsystem-sized testbench with multiple unit- and block-level components in EVM. A verification engineer was able to enroll the whole environment under EVM in less than half a day and add the EVM verbosity macros to two complex components as a proof-of-concept. The changed code has been run and tested in simulators of two of the three largest vendors. There were two challenges to be addressed:

1) If the verification component was not using an ARM verification base layer that was built upon UVM, several compilation runs were required to ensure that the EVM base layer has been applied to the class hierarchy only once. When the ARM verification base layer was made EVM-aware, no such issues have been observed.
2) In one of the simulators, the EVM base layer caused a compilation issue for a scoreboard that received its analysis implementation port (AP) type as a type parameter. The given instance used a not previously **typedef**-ed fixed-sized array of queues as the AP type. This seemed more like a simulator limitation as a **typedef** solved the error.

B. Application Notes

EVM proved to be a useful tool to automate reruns with changing verbosities during the simulation. As the provided API does not restrict when a verbosity descriptor is processed, it proved easy to set up specific trigger conditions related to simulation time or a specific event that elevated the verbosity of several components. With that, we were able to achieve significant speedups (up to two orders of magnitude in the chosen test case) as the simulations were able to run with a muted verbosity similar to **UVM_NONE** until the first event contributing to the error was reached and the range of interest of the debug started.

A simple trigger condition example could look like this:

```
//  Error is observed by the time of 701.3 us -> elevate verbosities at 699 us

//  ...existing code...

function void example_testcase::phase_started(uvm_phase phase);
  super.phase_started(phase);
  case (phase.get_name())
    "run": fork
      begin
        //  Wait 699 us after the start of run_phase()
        #699us;

        //  Load the debug verbosity configuration from the file evm_debug_config.json
        //  and update the verbosity values of all affected components
        `uvm_info(get_name(), "Parsing debug verbosity values from evm_debug_config.json...", UVM_NONE)
        evm_layer_globals::load_verbosity_descriptor("evm_debug_config.json");
      end
    join_none
  endcase
endfunction: phase_started


//  ...existing code...
```

VI. Conclusions

In this paper, we presented a methodology to overcome the complex task of UVM verbosity handling in testbenches of different vertical levels of integration. Although developed and evaluated in an internal ARM verification environment, the proposed methodology is tool-agnostic and applicable to any UVM-based verification flow. We demonstrated an implementation of EVM that proved to be a successful proof-of-concept regarding dynamic changes to verbosity settings in a cross-simulator form while preserving full backward compatibility with non-EVM-aware UVM components. We have produced evidence that it can be rolled out to larger-scale verification projects easily, and it does not interfere with a complex UVM-based class hierarchy and its **`uvm_factory`** overrides. Furthermore, it contributes to eliminating the creation of unnecessary messages and thus increasing simulation efficiency.

The externally stored verbosity descriptors proved to be an effective solution to increase debug turnaround times as this implementation eliminated the recompilation requirements. It has been shown that descriptors can be shared between unit- and integration-level environments and safely included in repositories. Although simulator-specific TCL-based verbosity controls offer flexible runtime control, they are often external to the verification codebase and difficult to reuse across environments. EVM addresses this limitation by providing a portable, source-controlled methodology for managing verbosity across vertical integration levels. The convenience and ease of deployment of this approach were acknowledged by design and verification engineers. While the current implementation seemed to be sufficient for daily use, we are currently exploring the inclusion of substring match-capable hashing algorithms to extend the capabilities of the verbosity descriptors.

We hope this work will be useful to verification engineers and will help encourage broader discussion of human–machine interaction in design verification.

References

[1] IEEE Std 1800.2-2020, IEEE Standard for Universal Verification Methodology

[2] J. Ridgeway, H. Nguyen, “Error injection in a subsystem level constrained random UVM testbench,” in Proc. DVCon US Conference, 2018

[3] IEEE Std 1800-2017, IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language

[4] S. Mellor, “User Programmable Targeted UVM Debug Verbosity Escalation,” in Proc. DVCon US Conference, 2025

[5] ibireme, “GitHub - ibireme/yyjson: The fastest JSON library in C,” GitHub, 2025. https://github.com/ibireme/yyjson (accessed Dec. 27, 2025)

[6] Cyan4973, “GitHub - Cyan4973/xxHash: Extremely fast non-cryptographic hash algorithm,” GitHub, 2023. https://github.com/Cyan4973/xxHash (accessed Dec. 27, 2025)

[7] GNU Project, “GNU Coreutils 9.8,” www.gnu.org, 2025. https://www.gnu.org/software/coreutils/manual/coreutils.html (accessed Dec. 27, 2025)