2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SANTA CLARA, CA, USA
MARCH 2 – 5, 2026

Enhanced Verbosity Methodology

Gergő Vékony, József Mózer

ARM Hungary

arm

Motivation

Why Talk About Verbosity?

Logs are our primary debug interface, as

- It captures the Stimulus (and Intent)
- It describes the Results (and Prediction)

While our tools evolve and grow, logs still dominate

- Logs are easy to search, No GUI is required
- UVM provides several ways of customization

Yet User Experience of working with logs is still not a concern

- Intelligent Search functions were added to tools
- AI and LLMs might be a solution?

Questions of Scale

Why Does it Break at Integration Level?

What is manageable on Unit level, might break on Integration level.

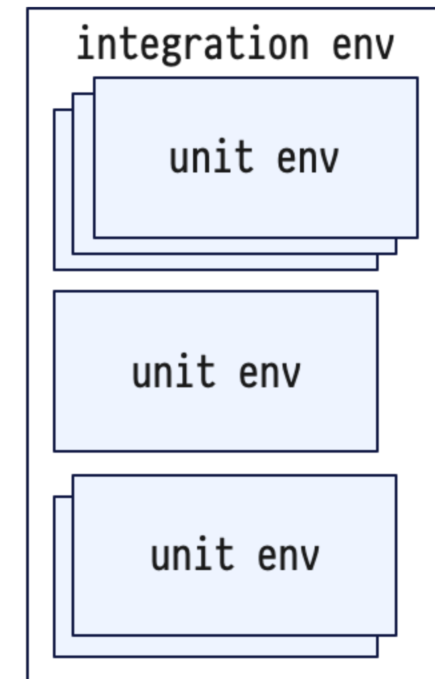
- Unit and integration level has different priorities
- Log and IO performance is rarely assessed on unit level

On Subsystem / System level, there might be

- Tens or hundreds of components instantiated
- Several instances of the same class used

Which often result

- Flooded logs in huge logfiles
- Suppressed information



What UVM Provides

Standard Verbosity Controls

UVM provides many options

- Global verbosity threshold
- Per-component verbosity threshold
- Report server customization
- Report catcher customization

DV engineers often use home brew solutions

And vendor specific solutions also exist

Problems of Practice

Real-World Limitations

Simulation performance is very dependent on log messages

- String handling (substitution, concatenation)
- I/O operations

Are always expensive

On higher integration levels

- More components ➜ more messages
- Components are often suppressed entirely

Design Goals

What EVM Tries to Fix

EVM implements

- Dynamic runtime verbosity control
- Filtering *before* message creation
- Several granularities for components
- Reusability across Unit, Subsystem and System level
- Backward compatibility with UVM
- Option for gradual rollout

What is EVM

Enhanced Verbosity Methodology

EVM is a lightweight UVM extension

- Orthogonal to report servers
- Orthogonal to report catchers
- Tool agnostic implementation

EVM does not replace UVM reporting!

Key Concepts
Dynamic Verbosity

Hard-coded verbosity ➜ verbosity *as data*

Verbosities are loaded from an external descriptor
- Default values are provided

Descriptor(s) can be loaded unlimited times
- Support for triggered escalation

Key Concepts
Message Categories

Message categories are introduced describe intent

- Build
- Header
- Detail
- Item (prints)
- Report

Each with their own specific verbosity value

Categorical Imperative

Why Categories Matter

Categories map to component lifecycle phases

- Build and connect messages are valuable during bring-up

Sometimes, it is more important to see

- Call flow than data
- Selective data than full items

Categories allow selective filtering or suppression

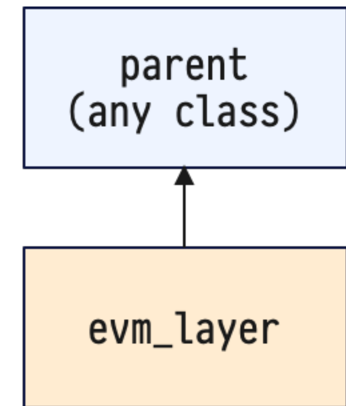
Key Technique

Opaque Class Extensions

Opaque class extensions

- Allow adding methods and variables to classes
- While the concrete implementation is not affected

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



2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Enabling EVM

Minimally Invasive, Gradual Rollout

To enable EVM,

- Class extension needs to be changed

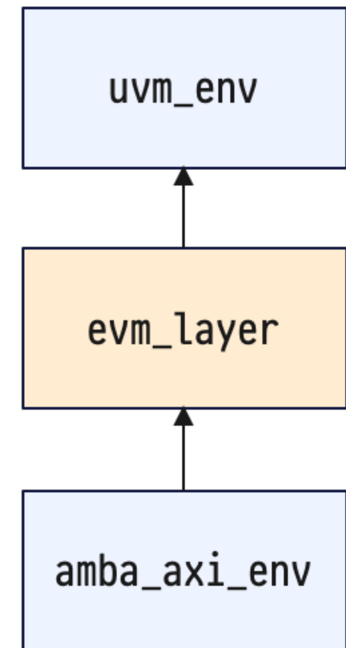
```
class amba_axi_env extends evm_layer #(uvm_env);
```

- Assign component group to map default values

```
function amba_axi_env::new(string name, uvm_component parent);
  super.new(name, parent);
  set_evm_group(evm_types::EVM_ENV);
```

- Ensure `super.build_phase()` is called

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



2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES SANTA CLARA, CA, USA MARCH 2 - 5, 2026

Verbosity Descriptors

Verbosity as Data

Verbosity values are stored in JSON

Scopes are provided for message categories of

- Group (role)
- Type (class)
- Instance (name)
- Full hierarchical name

JSON is plain-text ➜ version control

- Reusable across environments

Under the Hood
Performance Considerations

EVM is built for performance

- Local configuration object per component
- Central registries (hold config object *handles*)
  - Group
  - Type
  - Instance
  - Path
- Configuration objects identified by hashes
  - Strings are quirky!

EVM in Practice

Case Study Reports

EVM with an ARM SMMU Unit Level TB

- 100+ components were enrolled to EVM (used UVM)
- 2 checkers were modified to use EVM calls
- Test1 is a cache warmup, cache hit-miss testcase
- Test2 is a complex stress test with invalidations

The goal was to provide the same amount of log details in the 2 checkers with EVM as with UVM_HIGH.

Results are averages of 10 reruns:

- In Test1, an average of 18.5% runtime was consumed
- In Test2, an average of 9.4% runtime was consumed

Memory consumption reported 1.37Gb with and without EVM

Measurement Results

	No EVM	With EVM
Comp + Elab	3m 1s	3m 7s
Test1 UVM_NONE	11s	11s
Test1 UVM_LOW	24s	23s
Test1 UVM_HIGH	1m 21s	1m 22s
Test1 EVM		15s
Test2 UVM_NONE	50s	51s
Test2 UVM_LOW	1m 51s	1m 51s
Test2 UVM_HIGH	10m 57s	11m 3s
Test2 EVM		1m 2s

Conclusions

Why EVM is Useful

- It scales with integration level complexities
- Improves debug user experience
- Reduces unnecessary message creation
- Tool-agnostic, backward compatible implementation
- Practical for real DV teams
- Lightweight

Thank you!