CVM - A Library for Unified C++ and SystemVerilog Testbench Development

Varun Koyyalagunta, Jiahan Zhang, Mansoor Anees, Pravin Tavagad
Tenstorrent

***Abstract*- CVM is a novel utility library that bridges the gap between C++ and SystemVerilog in hardware verification environments. It provides a unified framework for communication, configuration management, logging, and topology handling across language boundaries. This paper presents the architecture and implementation of CVM, demonstrating how it enables more maintainable, reusable, and efficient testbenches for complex hardware designs. The library is slated for open-source release, offering the industry a standardized approach to mixed-language verification challenges.**

I. INTRODUCTION

Modern hardware verification faces increasing complexity as designs grow larger and verification requirements become more sophisticated. Traditional approaches often struggle with the integration of high-level algorithmic models written in C++ with the hardware-oriented SystemVerilog testbench infrastructure. While SystemVerilog excels at hardware interface manipulation and timing-accurate modelling, C++ provides superior performance for complex algorithms, reference models, and data processing tasks.

The industry lacks a standardized methodology for seamless C++/SystemVerilog integration. Existing solutions rely on low-level DPI-C interfaces that require extensive boilerplate code and careful memory management. This paper introduces CVM (C++ Verification Methodology), a comprehensive library that addresses these challenges through modern C++ techniques and systematic abstractions.

CVM's key innovation lies in its holistic approach to mixed-language verification. Rather than providing isolated solutions, it offers an integrated framework encompassing communication, component management, and code generation. The library leverages C++20 features, particularly coroutines, to enable intuitive asynchronous programming patterns while maintaining the performance characteristics required for emulation environments.

II. RELATED WORK

Hardware verification environments traditionally face challenges when integrating C++ models with SystemVerilog testbenches. Despite the widespread need, there is no standardized methodology for C++/SystemVerilog integration in the industry. Existing solutions like DPI-C provide basic communication but lack higher-level abstractions for structured data exchange, component discovery, and synchronized execution.

Universal Verification Methodology (UVM) primarily operates within SystemVerilog, offering limited guidance for C++ integration. Additionally, UVM's complex class hierarchy and runtime overhead make it poorly suited for emulation environments and high cycles-per-second (CPS) simulations, where performance is critical.

Python-based verification methodologies like Cocotb have emerged as alternatives, offering excellent productivity through high-level scripting capabilities. However, these frameworks suffer from significant performance limitations in high-throughput verification scenarios. The Python interpreter's overhead makes them impractical for emulation environments and high CPS simulations, where hardware-like execution speeds are required for meaningful verification.

Some proprietary methodologies exist within companies but remain inaccessible to the broader verification community. CVM fills this gap by providing an open-source solution that can be adopted as a standard across the industry, specifically designed for high-performance emulation compatibility while maintaining productivity benefits.

III. CVM ARCHITECTURE AND DESIGN

A. Overall Architecture

CVM consists of seven interconnected modules, each addressing specific aspects of mixed-language verification:

1. Messenger System: Publisher-subscriber communication with C++20 coroutines
2. Topology Management: Hierarchical component discovery and addressing
3. Packet Generation: Automated transaction type generation from YAML
4. Registry System: Topology-aware component lifecycle management
5. Plusargs Bridge: Unified command-line argument handling
6. Logging Framework: Cross-language structured logging
7. Utility Libraries: Bit manipulation and random number generation

The architecture follows a layered approach where low-level utilities support higher-level abstractions. The topology system provides the foundation for component addressing, enabling the messenger and registry systems to locate and manage distributed components efficiently.

B. Design Principles

CVM's design is guided by several key principles:

Type Safety: Template-based designs ensure compile-time type checking across language boundaries, eliminating common runtime errors in DPI interfaces.

Performance: Zero-overhead abstractions and efficient data structures maintain performance characteristics suitable for emulation environments.

Modularity: Each component can be adopted independently, allowing incremental integration into existing verification environments.

Modern C++: Leveraging C++20 features like coroutines, concepts, and ranges to provide intuitive programming interfaces.

IV. KEY TECHNICAL CONTRIBUTIONS

A. Messenger System with Coroutines

The messenger system implements a publisher-subscriber architecture that enables structured communication between C++ and SystemVerilog components. The key innovation is the integration of C++20 coroutines for asynchronous operations, allowing intuitive transaction-based communication without complex handshaking protocols.

B. Automated Packet Generation

The packet generation system eliminates error-prone manual DPI interface coding through automated code generation from YAML specifications. This approach ensures type safety across language boundaries while reducing development time.

C. Topology Management

The topology system provides hierarchical component discovery, allowing components to locate each other without hardcoded paths. This enhances reusability and test portability across different design configurations.

D. Registry System

The registry framework provides topology-aware C++ class instantiation that respects the design hierarchy. It automatically manages component lifecycles including construction, configuration, checking, and shutdown phases.

E. Cross-Language Utilities

The plusargs system provides unified command-line argument access from both languages using the VPI interface to bridge SystemVerilog simulator arguments to the C++ gflags library. This ensures consistent test configuration without duplicated argument parsing. The logging system provides structured output with verbosity control accessible from both languages. It includes automatic log rotation and custom handler registration for specialized logging requirements.

V. CASE STUDY: RISC-V PROCESSOR VERIFICATION

A. Application Context

CVM was successfully deployed in building a comprehensive testbench for a RISC-V processor core. The verification environment required co-simulation against an Instruction Set Simulator (ISS) reference model, implementing bit-accurate comparison of architectural state after each instruction execution.

The testbench architecture consisted of:

- SystemVerilog interface agents for instruction injection and result monitoring
- C++ ISS reference model for golden comparison
- C++ instruction generators for directed and random test creation
- Cross-language scoreboard for result comparison and coverage collection

B. Implementation Details

The topology system modeled the processor hierarchy, enabling automatic discovery of functional units and register files. The packet generation system created type-safe interfaces for instruction transactions and architectural state snapshots. The messenger system facilitated communication between the SystemVerilog testbench and C++ reference model using coroutines for non-blocking operation. This enabled the ISS to process instructions asynchronously while maintaining synchronization with the DUT.

C. Verification Benefits

The CVM-based approach provided several advantages over traditional methodologies:

- Reduced Development Time: Automated packet generation eliminated approximately 2000 lines of manual DPI interface code, reducing development time by an estimated 30%.
- Improved Maintainability: The topology-based component addressing enabled test reuse across different processor configurations without modification.
- Enhanced Debuggability: Unified logging provided consistent formatting across languages, reducing debug time for cross-language issues by providing trace correlation.
- Performance: The optimized communication mechanisms achieved performance within 5% of hand-tuned DPI implementations while providing significantly higher abstraction levels.

VI. PERFORMANCE ANALYSIS

A. Communication Overhead

Benchmarking of the messenger system shows minimal overhead compared to raw DPI calls. For typical transaction sizes (64-512 bytes), the overhead is less than 3% compared to optimized DPI implementations. The coroutine-based scheduling adds approximately 10ns per context switch on modern x86 processors.

B. Memory Utilization

The template-based design enables aggressive compiler optimization, resulting in minimal runtime overhead. Static analysis shows that CVM abstractions add less than 1% memory overhead compared to equivalent hand-coded implementations.

C. Emulation Compatibility

CVM has been verified to work correctly in emulation environments where performance is critical. The library's design avoids dynamic memory allocation in critical paths and provides configuration options to disable features that may impact emulation performance.

VII. INDUSTRY IMPACT AND ADOPTION

The open-source release of CVM addresses a significant gap in the verification industry. Early adoption feedback indicates strong interest from both commercial and academic institutions. The library's modular design enables incremental adoption, reducing barriers to entry for existing projects. Educational institutions have begun incorporating CVM into advanced verification courses, providing students with exposure to modern mixed-language verification techniques. This training pipeline will help establish CVM methodologies as industry best practices.

VIII. FUTURE ENHANCEMENTS

Several enhancements are planned for future CVM releases:

- **Domain Support**: Multi-domain verification with independent scheduling and synchronization points.
- **Distributed Verification**: Extensions to support verification across multiple processes or machines.
- **Language Bindings**: Additional language support including Python integration for scripting and analysis.
- **Formal Integration**: Hooks for formal verification tools to leverage CVM's component abstractions.
- **Cloud Integration**: Support for cloud-based verification environments with elastic resource management.

IX. CONCLUSIONS

CVM represents a significant advancement in hardware verification methodology by providing a comprehensive solution to C++/SystemVerilog integration challenges. It combines modern C++ features with practical verification needs, resulting in more maintainable and efficient testbenches. The successful application of CVM in verifying a complex RISC-V processor demonstrates its effectiveness in real-world scenarios. By standardizing the interface between C++ and SystemVerilog, CVM reduces the learning curve for new verification engineers and enables more efficient collaboration between team members with different language expertise.

The upcoming open-source release of CVM will allow the wider verification community to benefit from and contribute to this solution, potentially establishing it as an industry standard for mixed-language verification. The library's modular architecture and performance characteristics make it suitable for a wide range of verification applications, from academic research to large-scale commercial projects. As the verification industry continues to face increasing complexity challenges, solutions like CVM that combine high-level abstractions with performance optimization will become increasingly important. The foundation provided by CVM enables the verification community to focus on verification methodology innovation rather than infrastructure development.

ACKNOWLEDGMENTS

The authors would like to thank the open-source community for their contributions to the underlying technologies that made CVM possible, including the gflags library, the C++20 standard committee, and the SystemVerilog standards committee.

REFERENCES

[1] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2017, 2018.
[2] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2017, Direct Programming Interface (DPI), 2018.
[3] Universal Verification Methodology (UVM) 1.2 Class Reference. Accellera Systems Initiative, 2014.
[4] Cocotb: A coroutine based cosimulation library for writing VHDL and Verilog testbenches in Python. \url{https://cocotb.org}
[5] ISO/IEC 14882:2020 Information technology — Programming languages — C++. International Organization for Standardization, 2020.
[6] Google gflags library. \url{https://gflags.github.io/gflags/}
[7] Verilator: Open-source SystemVerilog simulator and lint system. \url{https://verilator.org}
[8] The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. RISC-V Foundation, 2019