Towards a Standardized Multi Language Verification Framework

First Prototype and Demonstration

MLV Working Group (Skeleton Subgroup)

Martin Barnasconi (NXP)
Alex Chudnovsky (Cadence)
Faris Khundakjie (Intel)
Bryan Sniderman (AMD)
Warren Stapleton (AMD)

accellera SYSTEMS INITIATIVE™

SYSTEMC™ EVOLUTION DAY
OCT 29, 2020 | VIRTUAL WORKSHOP

Copyright Permission

- A non-exclusive, irrevocable, royalty-free copyright permission is granted by **AMD, Cadence, Intel, NXP** to use this material in developing all future revisions and editions of the resulting draft and approved Accellera Systems Initiative **SystemC** standard, and in derivative works based on the standard.

MLVWG – Agenda

- Introduction (Warren Stapleton @ AMD – MLVWG Chair)
  - Problem Statement
  - Example Use Cases
  - High Level Architectural Requirements
  - Architecture
  - Example Use case
  - MLV API Standard
  - Sample API document
  - Software Architecture
- MLV Demo (Faris Khundakjie @ Intel – MLVWG – Co-chair)
  - Live demonstration
- SystemC Requirements (Martin Barnasconi @ NXP - MLVWG & Accellera TC Chair)
  - Learnings
  - Summary

MLVWG – Introduction

The mission of the MLVWG is to create a standard and functional reference for interoperability of multi-language verification environments and components.

- The MLVWG have reviewed requirements and are developing an open source proof-of-concept library for creating a standards-based approach for combining verification environments developed in different languages/frameworks.
- In addition, the group will look at ways to enable the introduction of UVM (Universal Verification Methodology) concepts in other environments and languages that come from legacy projects or developed with frameworks other than System Verilog or System C for beneficial reasons.

- Chair: Warren Stapleton (AMD)
- Vice Chair: Faris Khundakjie (Intel)

MLVWG – Problem Statement

- Verification engineers encounter multi-language (ML) integration problems on a regular basis.
- The ML integration problem is not limited to System Verilog⇔SystemC but can include models and stimulus in other languages such as VHDL, Matlab, e, C/C++, and interpreted languages.
- Many users share the same use cases and most (re)invest redundant efforts with non-standard internal or vendor-specific solutions.
- The problem grows when mixing technologies from different vendors for their unique benefits.

MLVWG – Example Use Cases

- Verification IP reuse
  - Combine VIP that has been written in different frameworks/languages into a coherent DV environment
- Software based stimulus
  - CPU/Embedded controller with own instruction set where UVM style sequences do not make sense.
- Reuse for simulation acceleration
  - SystemC reference and functional model reuse in hardware accelerated environments where a SystemVerilog testbench may not be present.
- Legacy code reuse
  - Seamless integration of legacy testbench code and utilities written in different languages.

MLVWG – Requirements Goals

- Goals
  - Standardize interoperability between multiple different verification frameworks through
    - communication between the frameworks
    - coordination with phasing/synchronization primitives
    - and making common methodology facilities available across the entire system.
  - Simplify the integration of external VIP
  - Deliver debug state and access methods for data that flows through the system.
  - Should also encompass emulation and hardware assisted verification environments.
  - Enable multiple implementations based on a standard.
- Tenants
  - Follow UVM principals and support a superset of the UVM functionality.

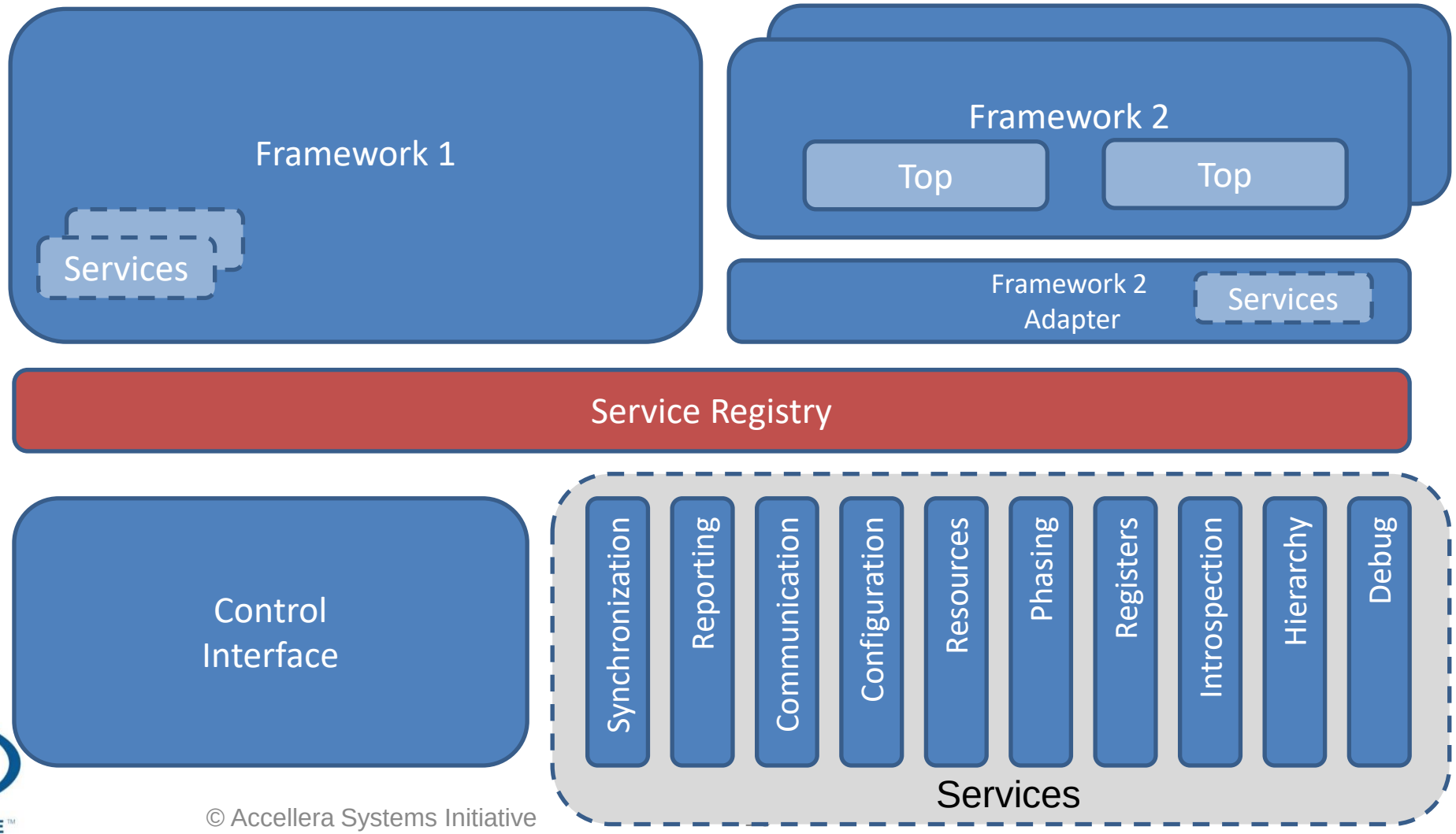
MLVWG Requirement Areas

Section	Description
System	Overall system level/architectural
Startup	Getting the ML system initialized
Shutdown	Cleaning up
Synchronization	Timing and event level coordination
Phasing	User level phasing
Communications	Sending data between frameworks
Control and Status	Controlling the system and determining what is going on
Messaging	Routing/processing user level messages and reporting
Debug	Figuring out what is going on across the ML system
Register Management	Access to Register meta-data from different frameworks
Compliance	Deliverables of the ML team and testing requirements

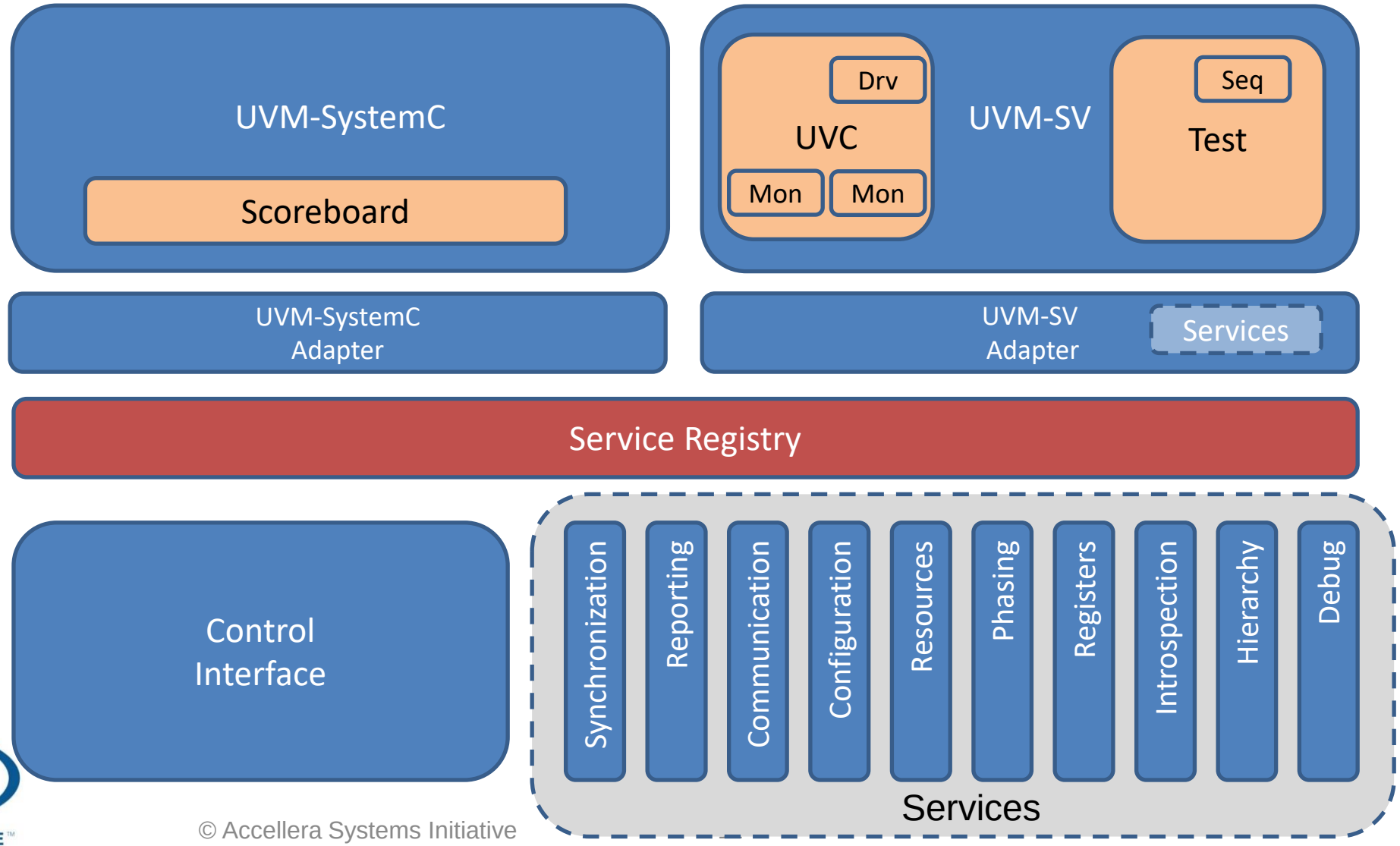
Guiding Principles

- Facilitate the creation of a scalable, modular and layered architecture
  - Following “Service-oriented framework” concepts
- MLV introduces adaptors that bridge frameworks that have UVM(-like) capabilities to the overall system
  - Phasing, configuration and communication offered as services.
  - Available services are selected using a resolution process.
- Decouple API from Implementation
  - Standardizing API (Documentation and header files)
  - Providing reference implementation.
  - Facilitate creation of one or more reference implementations.

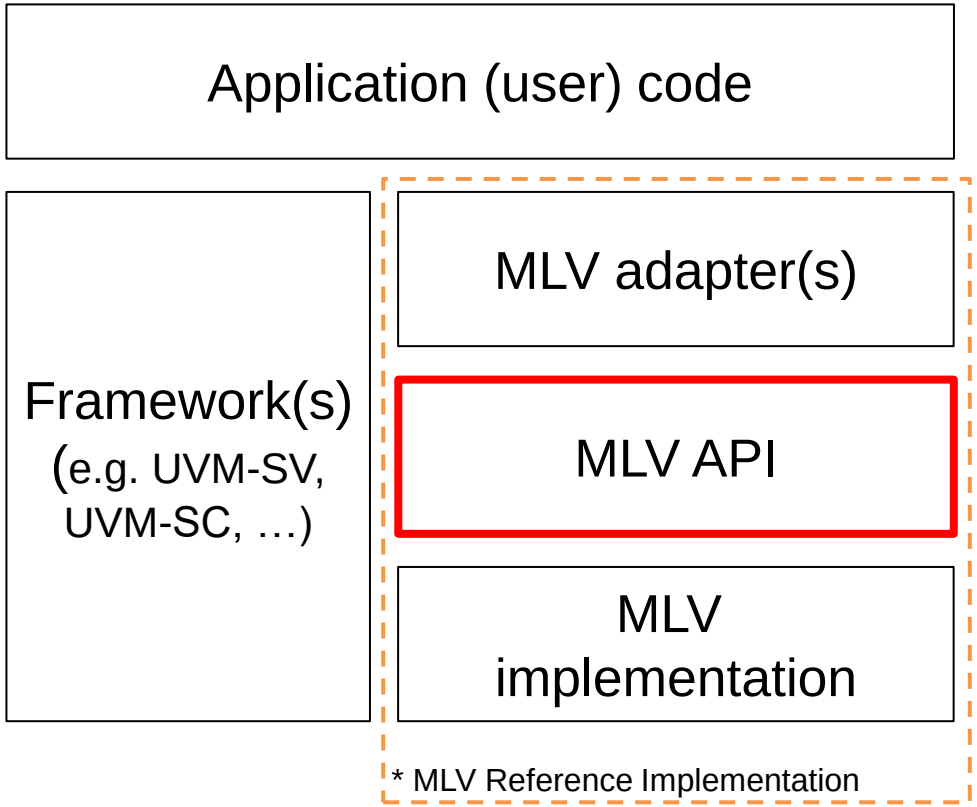
MLV Architecture



MLV Use Case: VIP Reuse



Software Architecture – layers



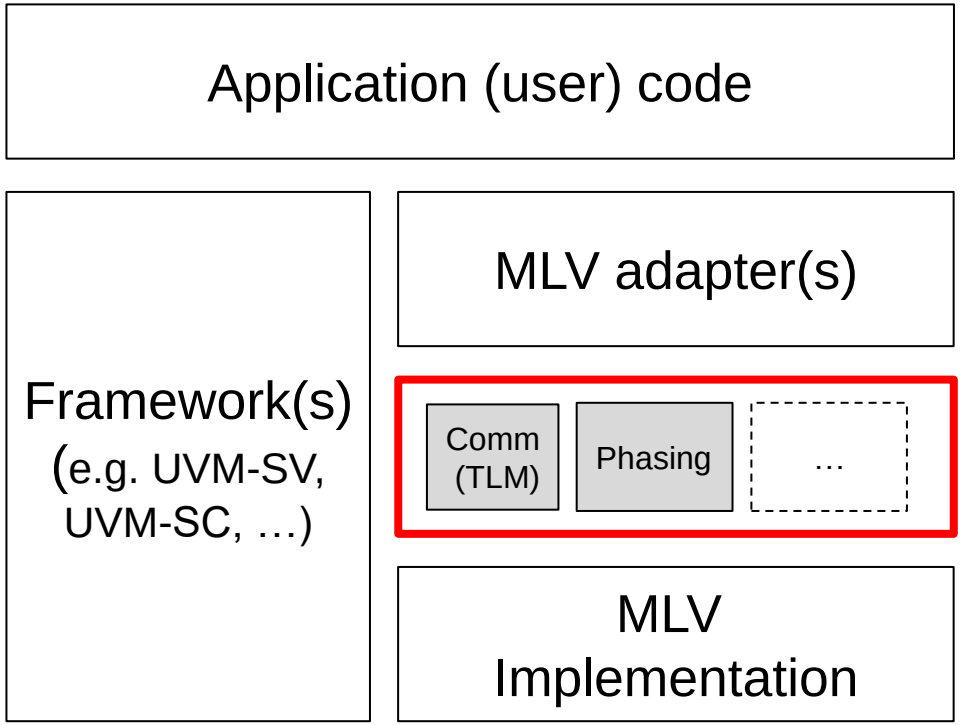
(Multi-language) verification environment developed by end-user

Multi-language framework adapters developed by EDA and/or Framework provider(s)

Standardized abstract interface definitions (“Accellera MLVWG standard”)

Implementation of the interfaces for services such as phasing, communication, etc.

Software Architecture – layers services

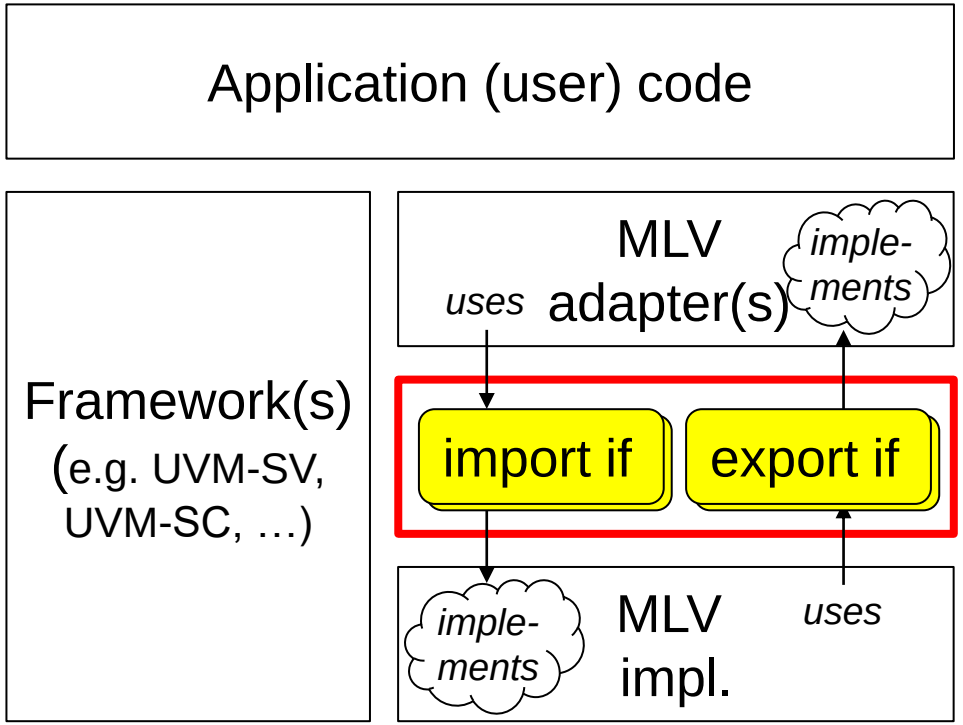


Initial focus of MLV API definition

- Lifecycle (Startup, Shutdown)
- Hierarchy
- Phasing

MLV API

Software Architecture - import/export



MLV API defines import/export interfaces (from adapter point-of-view)

Import interfaces

- *Use* the methods defined in the interface
- The method functionality (implementation) is owned by an MLV implementation

Export interfaces

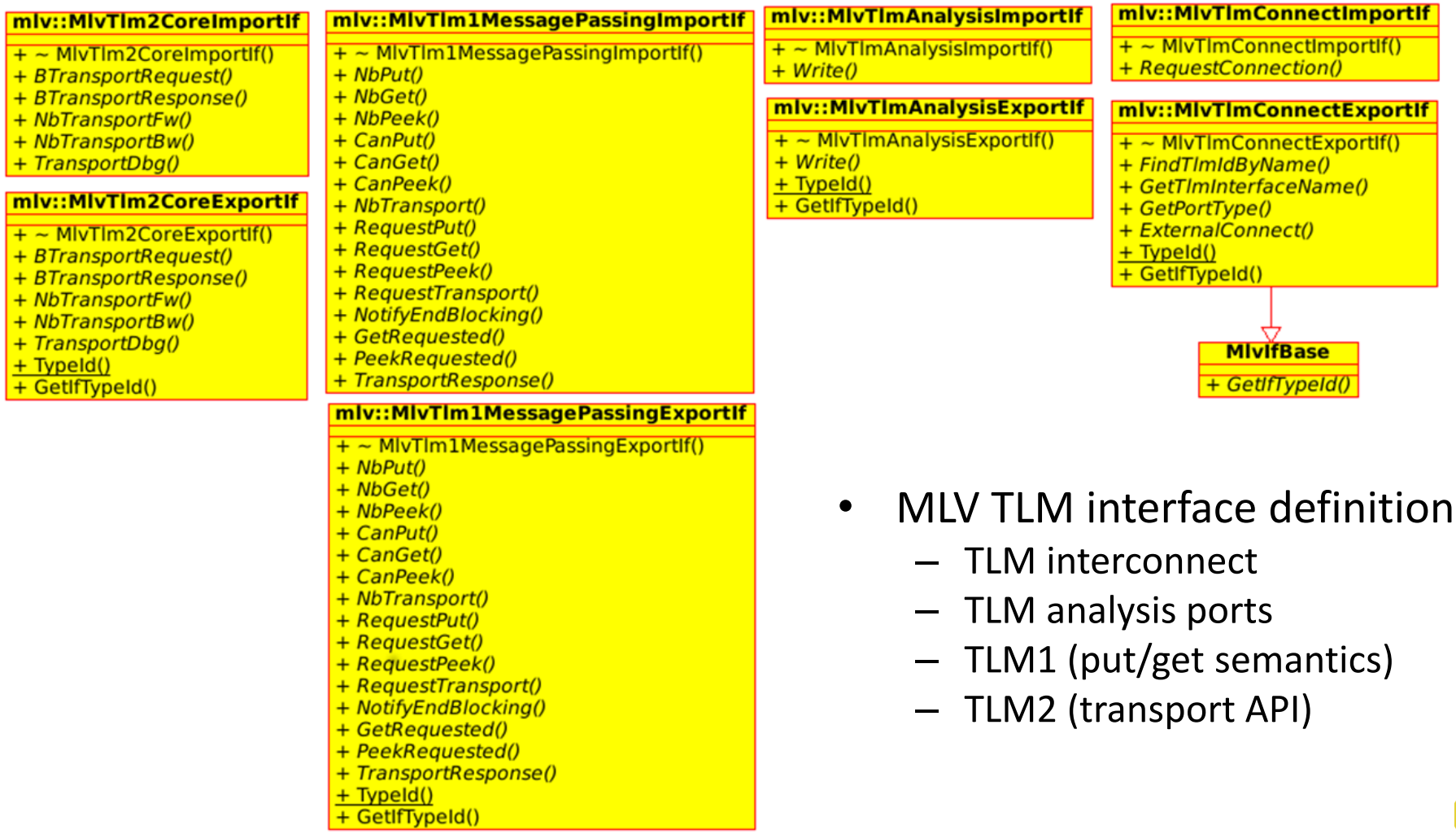
- *Provide* an implementation of the methods defined in an interface
- The method functionality is owned by the MLV adapter

API Naming Convention

- Introduced naming conventions to reflect ownership and direction of communication based on a popular C++ style guide.
- Pattern:
  `Mlv<Category><Actor-role><Direction><Type>`
  - Category: one of phasing, communication (TLM), configuration, etc.
  - Actor-role: Indicate role of the actor, e.g. Participant/Controller
  - Direction: Provide interface (Export) or apply interface (Import)
  - Type: Abstract interface type (`If`)

- Example:
  - `MlvPhaseParticipantExportIf`

Communication (TLM) API (Preliminary)



- MLV TLM interface definitions
  - TLM interconnect
  - TLM analysis ports
  - TLM1 (put/get semantics)
  - TLM2 (transport API)

MLV Demo

- Demonstrate basic MLV capabilities
  - Lifecycle
    - Startup
    - Shutdown
  - Phasing
- Live demonstration

SystemC multi-language requirements (1)

- Enable instantiation of verification components after elaboration of the design hierarchy, at the start of simulation
  - UVM-SystemVerilog instantiates verification components in the `initial` block, so at the start of simulation. Other frameworks should be able to interact with this structure. E.g. alignment between build and connect phases.
  - SystemC should introduce a dedicated phase to enable “late” instantiation of verification component **just before** simulation starts.
  - Initial proposal in SystemC github ticket #383 Introduce quasi static elaboration phase (MLVWG request)

SystemC multi-language requirements (2)

- Time-synchronization in a multi-language environment requires that SystemC stops at the specified time
  - Problem: `sc_start(<time>)` causes the SystemC kernel to stop **just before** the first delta of the specified <time>, rather than entering it
  - Proposal: `sc_start(<time>)` should stop at the **first delta** of the specified <time>
  - No SystemC github issue/ticket available yet

Other requirements

- Requirements related to UVM-SystemVerilog, UVM-SystemC standards and implementations:
  - The UVM environment shall perform the elaboration of the verification components by calling dedicated callbacks in the context of SystemC
  - Introduce an alternative methods to UVM `run_test` which decouples the creation of the verification environment from the actual start of the test
    - These methods are the entry point to start multi-language verification scenarios
    - Example: elaboration of the verification components `uvm_elab (<test>)`, Start of (ML) simulation: `uvm_start(<time>)`

Learnings & Findings

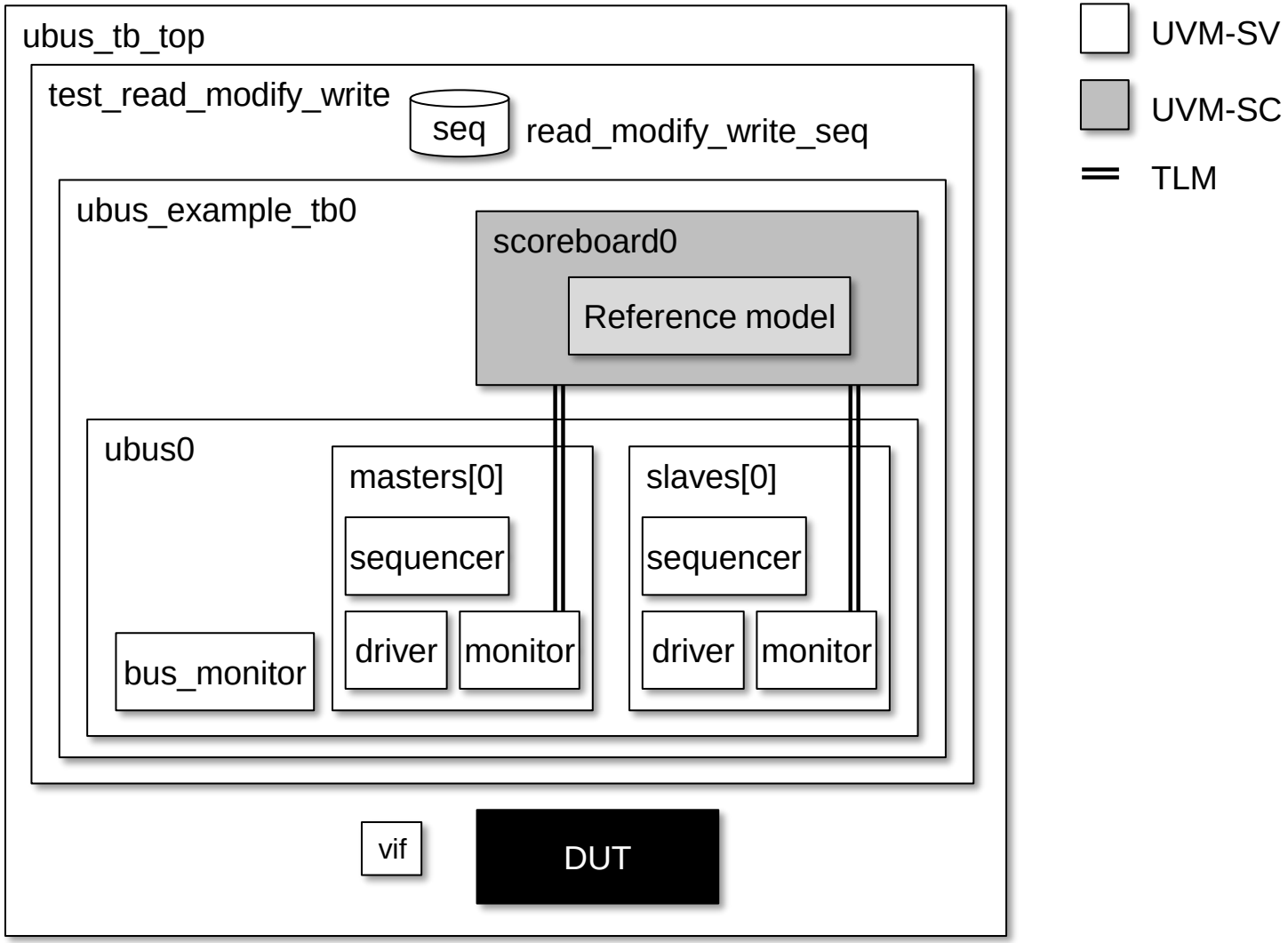
- We are dealing with a complex problem
  - Build upon capabilities and limitations of existing verification frameworks and standards (like UVM) or build something from scratch?
  - We learned how to leverage existing developments and technologies and adapt them to the MLV API / standard (e.g. UVM-ML)
- Practical challenge
  - Ownership on the adapter side regarding services that want to be used external to a framework
  - Introducing new capabilities without changing existing user code
  - Limited resources and time for API definition and implementation
- Strong commitment and contributions of Intel, AMD, NXP and Cadence
  - We can use your help … contact us if you'd like to contribute

Summary

- First API proposals are there
- Presented to SystemC group requirements for standardized extensions
- Have a demonstration using a partially implemented reference
- Next steps
  - Work on next area, TLM communication
  - Expand demo to include SV
  - Include more frameworks
  - Document API in language reference manual
- Future
  - Look for collaboration opportunities for other language framework adapters
- Join Us!
  - How to contribute - Contact the working group chair
    - mlwg-chair@lists.accellera.org

BACKUP

Example Use-case



MLVWG – Additional Requirements

- Dependencies (or more appropriately, *Independencies*)
  - Should not necessarily implement the functionality relying on a specific framework
  - User-level VIP code is not aware of the solution
  - Allow for future and proprietary verification frameworks to interoperate
- Control
  - Allow VIP in different frameworks to be controlled from different languages/frameworks
  - Provide the ability to forward external commands between frameworks
  - Allow for locality controls/filters to optimize messaging between frameworks*
- Access to commonly used facilities, e.g.
  - Deliver API to share access to commonly used features like memory managers and register abstraction layers*
  - Deliver visibility and control over when any framework is performing save-restore (serialization) actions