“In-emulator” UVM++ randomized testbenches for high performance functional verification

Adnan Hamid & David Kelf
Breker Verification Systems
Agenda

• UVM++ for efficient coverage closure

• UVM++ for fast IP emulation

• Enabling Firmware to run on UVM++ for IP simulation & emulation

• Reusing verification content for SoC System Coherency
The Verification Gap

**Block Functionality**
UVM simulation

**SoC Integration “Gap”**
Ad hoc test content

**System Validation**
Real-workloads on HW

- **UVM Testbench**
  - CPU
  - Memory
  - DMAC
  - AES
  - UART
  - VIP
  - System and Power Control

- **Bare Metal Firmware**
  - CPU
  - Memory
  - AES
  - Fabric
  - UART
  - VIP
  - System and Power Control

- **Simulation/Emulation Acceleration**

- **Hybrid Emulation Environment**

- **SW Drivers & OS**

---

**flexibility**

**performance**
The Verification Gap (IP and Sub-System)

- UVM is not scaling for complex IPs and sub-systems
  - UVM testbench & sequence development overshadows verification work

- Need emulation performance for IP verification
  - UVM testbench & sequence performance is limiting factor
  - Insufficient test content for sub-system testing

- Need firmware executing on IP simulation/emulation
  - Well ahead of when system is available
The Verification Gap (SoC and Post Silicon)

- Need reuse of IP / sub-system tests in SOC verification
- Need integration with System Coherency testing
  - cache coherency, power management, security etc.
Agenda

• UVM++ for efficient coverage closure

• UVM++ for fast IP emulation

• Enabling Firmware to run on UVM++ for IP simulation & emulation

• Reusing verification content for SoC System Coherency
What is UVM++?

- UVM Style API interface to PSS tool
  - Procedural SystemVerilog style classes
  - Also implemented in C/C++ for Firmware use

- Allows UVM experts easy access to PSS tool capabilities
  - No need to learn new language semantics

- Provides seamless integration to existing UVM testbenches
  - Coexists with existing test case, scoreboard etc.
Why UVM++ ? (The Problem)

- UVM is not scaling for complex IPs and sub-systems
- UVM testbench & sequence development overshadows verification work

High value verification knowledge...

... Locked up in uvm implementation complexity

Sequences
Stimulus Coverage
Scoreboard
Events & Scheduling
Control UVCs
“Constraint Hell”
Virtual Sequences
Debug

Interface VIPs
IP / Sub-System RTL

UVM Environment
Why UVM++ (The Solution)

- High value verification content captured in portable model
- Synthesize self-checking test from UVM++ graph-based models
- Synthesizable VerificationOS maps content to existing UVM testbench
action aes_encrypt {
    input buf in;
    input buf key;
    output buf out;
    lock aes_r lock;

    // Start of user code Action_aes_encrypt
    constraint in.len == 16;
    constraint key.len == 16;
    constraint out.len == 16;
    ref aes_regs regs;

    void post_solve() {
        in.addr = regs.AES_INPUT0.get_address();
        key.addr = regs.AES_KEY0.get_address();
        out.addr = regs.AES_OUTPUT0.get_address();
    }

    void body() {
        pss_info( name(), "aes_encrypt", pss::target );
        regs.AES_CTRL.START.set(1);
        regs.AES_CTRL.MODE.set(0); // 0 for Encrypt, 1 for Decrypt
        regs.AES_CTRL.write();
        regs.AES_CTRL.DONE.poll(1); // wait for completion

        // call reference model to predict results
        encrypt_aes ( in.expect, key.expect);
    // forward expect to output
        out.expect = in.expect;
    }

    // End of user code
}

// End of user code

// Generated Model

// Start of user code Action_aes_enc_dec

action aes_encrypt {
    input buf in;
    input buf key;
    output buf out;
}
Fitting UVM++ content into an existing UVM IP testbench

UVM sequence detail to interface w/ VIPs

```verilog
class trek_apb_master_seq extends uvm_sequence #(apb_pkg::apb_transfer);
  `uvm_object_utils(trek_apb_master_seq)

  virtual task body();
  forever begin
    req.get(m_tb_path, trek_done);
    if (trek_done) break;
    `uvm_send(req)
    if (req.direction == APB_READ) begin
      rsp.send(m_tb_path);
    end
    req.item_done(m_tb_path);
  end
endclass
```

UVM environment

- DPI_LIB := ${BREKER_HOME}/build/lib/libtrek.so
- INCDIR += ${BREKER_HOME}/target/sv
- VLOGSRC += ${BREKER_HOME}/target/sv/trek_uvm.sv
- SIM_OPTION += +TREK_TBX_FILE=test.tbx
Running a single IP test
Composing UVM++ IP models

Generated Model

```cpp
action dmac_xfr {
  input buf in;
  output buf out;
  lock dmac_r lock;

  // Start of user code Action_dmac_xfr
  constraint in.len == out.len;
  ref dmac_regs regs;

  void body() {
    int chan = lock.instance_id;
    pss_info (name(), "dma_xfr", pss::target);
    // configure target and source addrs
    regs.dma[chan].DMA_TADDR.ADDRESS.set(out.addr);
    regs.dma[chan].DMA_SADDR.ADDRESS.set(in.addr);
    regs.dma[chan].DMA_SADDR.write();
    regs.dma[chan].DMA_BUFF.SRC_INCR.set(1);
    regs.dma[chan].DMA_BUFF.DST_INCR.set(1);
    regs.dma[chan].DMA_BUFF.write();
    // start transfer
    regs.dma[chan].DMA_TRANS.SIZE.set(in.len);
    regs.dma[chan].DMA_TRANS.START.set(1);
    regs.dma[chan].DMA_TRANS.write();
    // wait for completion
    regs.dma[chan].DMA_INT_STATUS.COMPLETED.poll(1);
    // forward expect data
    out.expect = in.expect;
  }
  // End of user code
};
```
Fitting UVM++ into existing UVM Sub-system testbench

UVM sequence detail to interface w/ VIPs

```vhd
class trek_axi_master_seq extends uvm_sequence #(axi_pkg::axi_transfer);
  `uvm_object_utils(trek_axi_master_seq)

  virtual task body();
  forever begin
    req.get(m_tb_path, trek_done);
    if (trek_done) break;
    `uvm_send(req)
    if (req.direction == AXI_READ) begin
      rsp.send(m_tb_path);
    end
    req.item_done(m_tb_path);
  end
endclass
```

UVM environment

- DPI_LIB := ${BREKER_HOME}/build/lib/libtrek.so
- INCDIR += ${BREKER_HOME}/target/sv
- VLOGSRC += ${BREKER_HOME}/target/sv/trek_uvm.sv
- SIM_OPTION += +TREK_TBX_FILE=test.tbx

UVM Testbench

- SS RTL
- VIP
- Memory
  - DMAC
  - AES
- UART1
- UART0
- System and Power Control
- VIP

Testbench connections:

- SS Scenario Model
- AES
- UART0
- DMAC
- UART1
- BREKER TrekGen
- test.tbx

ACCELLERA SYSTEMS INITIATIVE
Running multi-IP sus-system test
3D Coverage Closure

Combinatorial
Function been tested?
Sequential
Sequence of functions been tested?
Concurrent
Sequence of sets over time

Parallel sequence threads
Concurrent
Has this timing combination been tested?
Sequential

Path Constraint/Coverage

SD Card Controller
Read
Display Controller

Photo Processor
Decode

Camera

Photo Processor
Encode

SD Card Controller
Write

SD Card Controller
Read
Photo Processor
Decode
Display Controller

Camera
Photo Processor
Encode

SD Card Controller
Write

Path Constraint/Coverage

Sequential

Thread 1
OR
Port/Reg 1
SD Card Controller
Read
Photo Processor
Decode
Display Controller

Camera

Test

Thread 2
OR
Port/Reg 2
SD Card Controller
Read
Photo Processor
Decode
Display Controller

Camera

Thread 3
OR
Port/Reg 3
SD Card Controller
Read
Photo Processor
Decode
Display Controller

Camera

Path Constraint/Coverage
High Level Scenario Debug

4 processors, 1 thread per processor

Green nodes have finished

Yellow nodes are running

Memory Usage by Region

Blue nodes are waiting to run

Failing Node

Test Execution Log

Data used/changed by a task

Software Driven Test Source

Tight integration to Verdi
Pre-generation, Reactive and Hybrid constructs

• Prefer pre-generation constructs
  • Better emulation performance
  • Simpler reuse in SoC and post-silicon
  • Supports checks, polls, scheduling etc.

• Prefer full reactive constructs for block level verification
  • Necessary when DUT response cannot be predicted

• Hybrid mode allows most operations to be pre-generated, with reactive generation where needed
  • Small sacrifice in simplicity and performance for flexibility
Functional, UVM++ Test Content

- VerificationOS provides a “layer”
- Under the layer the UVM testbench remains the same, with connections based on RAL.
- Above the layer, the UVM++ tests can be generated and applied
- This allows the same tests to be ported as the design scales
Agenda

• UVM++ for efficient coverage closure

• UVM++ for fast IP emulation

• Enabling Firmware to run on UVM++ for IP simulation & emulation

• Reusing verification content for SoC System Coherency
Emulation Functional Test Use Model Issues

- Not enough tests to take advantage of emulation capacity
- Long compile loop at odds with frequent test recompile
- Integrated simulator for random solver slows performance significantly
- Coverage and debug data collection slows performance
Pre-Execution, Coverage-Driven, Randomized Test Generation: Preserving Emulation Performance

Randomization on spec model, prior to execution
No testbench simulator

Coverage performed up-front, on model

Synthesized tests loaded straight into memory – no lengthy compile

Low to no data dump required for coverage and debug
Fitting UVM++ into IP Emulation Simulation Acceleration

C/C++ sequence detail to interface w/ AVIPs

```c
void run(){
  bool trek_done = false;
  while(1){
    req.get(m_tb_path, trek_done);
    if (trek_done) break;
    drive_transactor( req, rsp);
    if (req.direction == APB_READ){
      rsp.send(m_tb_path);
    }
    req.item_done( m_tb_path );
  }
}
```
Randomization Up-Front

- All randomization and scheduling solved off-line
- Allows emulator transactor to be driven at speed
Eliminating Test Content Compile

• TrekBox loads generated schedule at runtime

• No need for test compiles
Handling Coverage Up-Front

- Coverage reachability and coverage analysis available before tests are run
- Review coverage to decide if test suite should run
- No need to track coverage data at run time, leading to better performance
Debug Data Minimization

- Synthesizable VerificationOS collects minimal debug data
- Most debug information already available in generated schedule
Portable UVM++ test can be applied to emulator

• The VerificationOS layer also allows the UVM++ tests to be ported to the emulator

• UVM++ enables full pre-execution randomization, compile bypass, etc.

• Emulation performance is maximized
Agenda

• UVM++ for efficient coverage closure

• UVM++ for fast IP emulation

• Enabling Firmware to run on UVM++ for IP simulation & emulation

• Reusing verification content for SoC System Coherency
Running Firmware Without an OS or a Processor

• For firmware execution on a subsystem, how can we connect the registers in SW and HW?
• How can we provide the services needed by firmware, such as memory allocation?
• How can we load the firmware into the device memory?
Implement Firmware HAL in UVM++

#include “registers.h”

```c
void aes_encrypt(uint64_t offset){
    REG_WRITE(offset + reg_AES_CTRL, AES_CTRL_START & ~AES_CTRL_MODE);
    REG_POLL(offset+ reg_AES_CTRL, AES_CTRL_MODE, AES_CTRL_MODE);
}
```

```c
#define REG_WRITE( addr, value )
   \   trek::reg r = regs.get_reg_by_addr(addr);
   \   r.write(value);

#define REG_POLL( addr, value, mask )
   \   trek::reg r = regs.get_reg_by_addr(addr);
   \   r.poll(value, mask);
```
Leveraging a Light VerificationOS for Firmware

- VerificationOS provides enough OS capabilities while avoiding slow Linux bootup and performance
- HW Registers defined in UVM RAL layer, SW registers in headers
- Memory allocation, interrupt processing and IO transactions also operated by OS
- OS schedules operations, provides mapping, and synchronizes C with IO
Synthesizable VerificationOS Requirements

- Task & Resource Scheduling
- Execution Management
- Virtualized OS Services

Vertical Reuse
- System Synthesis
- Sub-System Block / IP Synthesis
- Virtual P. SoC
- Sub-System Block / IP
- SoC
- Horizontal Reuse
- Simulation
- Emulation
- Prototyping
- Post-Silicon

Gen-Time
- Testbench
- VIP
- Fabric
- Processor
- Controller
- SD Card
- CPU
- Memory
- Display
- Photo
- Testbench
- System
- Power and Sub-System Block / IP

Virtualized OS Services Mapping
- Memory Region 1
- Memory Region 2
- Memory Region 3
- Raw Image #1
- JPEG Image #2
- Coverage
- Debug, Profiling
- Multi-threaded Tests in SW

Run-Time Execution Management
- Synchronization

Synthesizable VerificationOS Services
- Messages
- Memory
- Registers
- Transaction
- Interrupts

Multi-processor
- Decode
- Encode
- Time
- DC
- JPEG
- Cover

Image #1
- JPEG
- Controller
- SD Card
- Camera
- Display
- Memory
- Testbench

Image #2
- JPEG
- Controller
- SD Card
- Camera
- Display
- Memory
- Testbench
Firmware at the Block, Sub-System & Full SoC
Agenda

• UVM++ for efficient coverage closure

• UVM++ for fast IP emulation

• Enabling Firmware to run on UVM++ for IP simulation & emulation

• Reusing verification content for SoC System Coherency
Reusable SoC System Coherency VIP Library

- SoC infrastructure testing can be effectively handled using pre-built scenarios that can be configured for the design
- By composing these at the specification abstraction level and synthesizing them, we can target coverage levels and corner cases not possible using templating
- Breker has a library, and other users create their own over time
TrekApps may be Configured and Expanded

Example Customizations
Specific Component Characteristic
Special Coherency Test Algorithm
Extra Processor Instruction
Combining TrekApps with UVM++

- Randomization on spec model, prior to execution
- No testbench simulator
- Coverage performed up-front, on model
- Synthesized tests loaded straight into memory – no lengthy compile
- Low to no data dump required for coverage and debug
Summary

• UVM++ for efficient coverage closure

• UVM++ for fast IP emulation

• Enabling Firmware to run on UVM++ for IP simulation & emulation

• Reusing verification content for SoC
Thanks for Listening!
Any Questions?