Creating Portable Stimulus Models with the Upcoming Accellera Standard

Part 1
- Portable Stimulus: The Next Leap in Verification & Validation Productivity
- Introducing Portable Stimulus Concepts & Constructs
David Brownell, Analog Devices Inc.
Faris Khundakjie, Intel

PORTABLE STIMULUS
THE NEXT LEAP IN
VERIFICATION & VALIDATION PRODUCTIVITY
A Brief History of Verification

Verification Productivity

Verification?: Directed Testing

Those who can, Design; those who can’t, Verify

System Verilog
CRV & MDV

Standard Methodology
UVM
VMM
OVM

Portable Stimulus
Why Portable Stimulus?

Verification Productivity is not scaling with complexity of projects.

- Need to reduce product life cycle w/ efficiency gains via portable content
  - Use cases replicated in different stimulus languages on different execution platforms

- Enforce single interpretation of a product specification
  - Disjoint activities in different platforms lead to expressing, covering & debugging multiple times

- Enable mainstream and methodical automation of test content reuse
  - Reuse between IP and SoC is a significant challenge in pre-silicon simulations
  - Technology advantages of different platforms not utilized efficiently to reduce investment

- Encourage verification and validation plans to become a continuum
  - Precious verification and validation knowledge not captured and reused
  - Test planning activities mostly disjoint, escapes to later stages b/c of earlier assumptions
Reimagined Development Process

PS Enabled Development Framework

- Architect
  - Use Cases, PSS Models

- IP Design/DV
  - IP DUT, PSS Models

- System Design/DV
  - SUT, PSS Models

- Apps Eval
  - Scenarios

- Software
  - Device Drivers

- Test
  - Scenarios

- TLM Model
  - SystemC

- Simulation
  - SV, UVM, C/C++

- Evaluation, Customer Boards
  - C/C++

- Virtual, Emulator, Silicon
  - C/C++

- Tester
  - Multiple
What Portable Stimulus Is and Is Not

**Portable Stimulus is:**
- A single representation of test intent that is reusable:
  - By a variety of users
    - Architects, Validation, DV, Test, Software
  - Across different levels of integration
  - In a variety of execution platforms
    - Post-Si, FPGA-prototyping, Virtual, Emulation, Simulation and more
  - Under different configurations within and across the dimensions above

**Portable Stimulus is not:**
- One forced level of abstraction → Expressing intent from different perspectives is a primary goal.
- Monolithic → Representations would typically be composed of portable parts.
- Intended to replace all testing activities in any single platform.
What About UVM?

The Good
- Common Language/Framework for Verification Engineers
- Smart Testbench Architecture to allow for “Checkers” to be reused vertically

The Bad
- Non DV & Designer Engineers are not familiar with SystemVerilog & UVM
  - Overly complicated and hard to debug
  - Need to be an expert in UVM to create a simple directed test

- Excellent for Block/IP Level Verification
  - Does not scale to System Level Verification, Only Solves “Checking” Portability Problem
  - Stimulus at block level SV Based, at system level needs to be C code
  - Methodology does not translate seamlessly from simulation to emulation

- No consideration for Software/Evaluation/Test Engineering needs
## What is the PSWG Trying to Fix?

### Product Development Cycle

<table>
<thead>
<tr>
<th>Role</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architect</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apps</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eval</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Test</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Focus</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer Scenarios</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Micro Architecture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eval</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drivers Tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functionality Performance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Everything</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Platform</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTL Schematic</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Emulator Silicon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluation Board</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tester</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verilog VHDL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Portability Use Cases & Potential Capabilities

<table>
<thead>
<tr>
<th>Stimulus Reuse</th>
<th>Simplified Test Authoring</th>
<th>Improved Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertical Reuse</td>
<td>Verification Stimulus Libraries</td>
<td>Test Visualization, Clearer Communication</td>
</tr>
<tr>
<td>IP → System</td>
<td>Coverage Based Test Creation</td>
<td>Improved Debug Efficiency</td>
</tr>
<tr>
<td>Horizontal Reuse</td>
<td>Dataflow Based Test Creation</td>
<td>Enable Customer/Vendor Innovations!</td>
</tr>
<tr>
<td>Simulation → Emulation</td>
<td>Multiple Constraint Types</td>
<td>Manufacturing Tests?</td>
</tr>
<tr>
<td>Simulation → Silicon</td>
<td></td>
<td>Formal?</td>
</tr>
<tr>
<td>Project A → Project B</td>
<td></td>
<td>Machine Learning?</td>
</tr>
<tr>
<td>Bi-Directional Reuse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eval Board Failure to IP Test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reuse SW Drivers in Simulation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Simplify Test Authoring

### Different Roles have Different “Care-Abouts” & “Points of View”

<table>
<thead>
<tr>
<th>Role</th>
<th>Care-Abouts</th>
<th>Points of View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architect</td>
<td>Throughput, Latency, QoS</td>
<td><strong>Want to write tests from the System Point of View</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enable All DMAs in parallel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Create test where DMA FIFO full while core in deepsleep</td>
</tr>
<tr>
<td>Block/IP Dsgn/DV</td>
<td>Micro-Architecture Functionality Performance</td>
<td><strong>Want to write tests that are IP Centric</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exercise all modes, error conditions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Then Create random combinations of those modes</td>
</tr>
<tr>
<td>System Dsgn/DV</td>
<td>Correct Connectivity Use Cases System Robustness</td>
<td><strong>Want to write tests from System Point of View</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interrupts/Pinmux/Fabric properly connected to IP Block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tests to ensure all memory regions accessible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Complex Stress Tests</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;Real&quot; system-level use case tests</td>
</tr>
</tbody>
</table>

### Existing languages are not expressive enough

### Existing runtime frameworks are not portable
Deployment Use Cases: Transactional Reuse

IP Use Case (e.g. flow of register writes and polls)

PSS Tool

UVM RAL Sequences

UVM Target Driver

IP Simulation - BFM

SoC Simulation - BFM

IP Use Case (e.g. flow of register writes and polls)

PSS Tool

UVM RAL Sequences

UVM Target Driver

Sync

C-test
Memory-mapped
register access

SoC Simulation - Core

SoC Simulation - Core
Deployment Use Cases: SW Driver Prototype

1. IP SW Driver Prototype in Portable Stimulus
   - PSS Tool
   - Short UVM RAL Sequences
     - UVM Target Driver
     - IP Simulation - BFM

2. IP SW Driver Prototype in Portable Stimulus
   - PSS Tool
   - Med length driver prototype in C
     - IP FPGA – Core
     - SoC Emulation – Core

3. IP SW Driver Prototype in Portable Stimulus
   - PSS Tool
   - Large length driver prototype in C
     - Post-Si Tester – Core
Deployment Use Cases: SW and FW Awareness

- Abstract memory allocation in Portable Stimulus
  - PSS Tool
  - C lib with linker managed memory
    - SoC Emulation – Linux OS

- Abstract memory allocation in Portable Stimulus
  - PSS Tool
  - Transactions in C with PSS managed memory
    - SoC Emulation – no Operating System
Why a Dedicated PS Standard?

- Enable Expansion of VIP Ecosystem → beyond UVM simulation VIP
  - Accelerate SoC integration/testing across all platforms via EDA PS libraries
  - Empower portable compliance testing suites by protocol standards (PCIe, etc.)

- Enable innovation for reuse across platforms from EDA vendors
  - A standard levels playing field and focus innovation on next set of challenges

- Increase predictability for mobility across platforms and vendors
  - Emulation & Simulation EDA suppliers may be different
  - Standard enables common input for both platforms
  - Standard dictates consistent semantics of user input and experience across execution platforms
PSS – No Reference Implementation

- Unlike some Accellera standards (e.g. UVM and SystemC), PSS does not include any reference implementation
  - Large number of possible implementations and techniques across platforms
  - Leave room to foster innovation across vendors and technologies as applications mature

- Specification and supporting material (tutorials, training, etc.) do not describe how any vendor implementation works
  - Vendor implementation is responsible for processing user input to generate and execute tests that interact with user execution environments
  - Questions on particular vendor implementation (aka secret sauce or PSS tool) should be discussed directly between a user and a vendor
  - Best effort is made in examples to help the user community visualize possibilities or modes of user portable input interactions with user execution environments
PSS – Two Possible Input Formats

PSS defines new fundamental semantics
- No existing common language (general purpose or other) specifications and associated compilers satisfy all current and future portable stimulus semantics
  - Scheduling Rules, Resource Assignment Rules, etc.
- Example: SystemVerilog scheduling regions specification was created and optimized with simulators in mind
- Example: Loops and memory allocations cannot be left to C++ common compilers; they need to be controlled by PSS tool depending on platform

Balancing need for input optimized for expressing portable stimulus against users familiarity with popular languages across platforms
- PSS development effort addresses need to have a natural and concise way for users to define portable input
  => Result is a Domain Specific Language (DSL) to express portable input
- Most platforms outside simulation use a few popular languages including C, C++ and Python for writing tests (particularly SoC tests)
  => Result is a C++ Input, with semantics matching DSL, also consumed by a PSS tool
INTRODUCING PORTABLE STIMULUS CONCEPTS & CONSTRUCTS

Tom Fitzpatrick, Mentor Graphics Corp.
Raising the Abstraction Level

- **Gate → RTL => Synthesis**
  - Randomize numbers ($urandom(), etc)

- **RTL → Transaction => UVM**
  - Constrained random transactions (randomize())
  - Structural randomization/customization (build(), config_db())

- **Transaction → Scenario => Portable Stimulus**
  - Declarative partial specification of key intent
  - Randomize scenarios based on system-level constraints
Stimulus at a Higher Level

Transaction-Level

class my_seq extends uvm_sequence#(tr_t);
`uvm_object_utils(my_seq)
task body;
    for(int i=0; i< 3; i++) begin
        req = tr_t::type_id::create("tr");
        start_item(req);
        req.randomize() with {...};
        finish_item(req);
    end
endtask
endclass

Scenario-Level
Actions Capture Intent

- Behaviors captured as actions
  - Simple actions map directly to target implementation
  - Compound actions modeled via activity graphs

- Actions are modular
  - Reusable
  - Interact with other actions
  - Inputs and Outputs define dataflow requirements
  - Claim system resources subject to target constraints

- Activity graph defines scheduling of critical actions
  - Define scheduling constraints
  - Flow objects and resources constrain scenarios

```c
void test() {
    // modify_translation_seq
    do_stw(0x104, 0xA4ADD4A4); 
    do_stw(0x110, 0xB4BBBBBB); 
    do_switch_translation(...);
    do_load_check_word(1); 
    // modify_translation_seq
    do_stw(0x112, 0xA4ADD4A4); 
    do_stw(0x140, 0xB4BBBBBB); 
    do_switch_translation(...);
    do_load_check_word(1); 
    // modify_translation_seq
    do_stw(0x120, 0xA4ADD4A4); 
    do_stw(0x102, 0xB4BBBBBB); 
    do_switch_translation(...);
    do_load_check_word(1);
}
```
Simple Example: UART

- UART receives/transmits data packets via Data port
- Packets DMA to/from memory
  - Concurrent with receive/transmit operation
- Command port accesses registers/memory
  - Configure UART/DMA
  - Read/Write MEM data

```plaintext
stream data_stream_s {
  rand int size;
}

buffer data_buff_b {
  rand int size;
}
```
Simple Example: UART

```
stream data_stream_s {
    rand int size;
}
buffer data_buff_b {
    rand int size;
}
```

```
action q2m_xfer {
    input data_stream_s src;
    output data_buff_b dst;
}
```

```
action m2q_xfer {
    input data_buff_b src;
    output data_stream_s dst;
}
```

```
action read_in_a {
    output data_stream_s data;
}
```

```
action write_out_a {
    input data_stream_s data;
}
```

Generate pkt stream

DMA pkt stream into mem buffer

DMA mem buffer into pkt stream

Consume pkt stream
Actions are *Modular*

- **Behavioral descriptions can be reused**
  - Behaviors eventually mapped to VIP and/or embedded CPU in target
  - Flexible mechanism to map to different targets

- **Actions encapsulate**
  - Their own intrinsic properties
  - Rules for interaction with other actions

- **Actions represent functionality**
  - First step is to identify target design behaviors to be exercised
  - What data do these behaviors require/produce?
  - Where are these behaviors executed? (DUT? VIP?)
  - What system resources are required to accomplish these behaviors?
  - These questions are independent of the implementation details of the DUT
action data_rx {

    read_in_a rd_i;
    q2m_xfer q2m;

    constraint {q2m.src.size % 4 == 0;}

    bind rd_i.data q2m.src;

    activity {
        parallel {
            rd_i; //receive data in UART
            q2m; //DMA into memory
        }
    }
}
Basic Scenario – Data Receive C++

Declare action

Declare subactions

Declare constraints
Constraints must be named

Bind subaction inputs/outputs

*activity* defines the graph of subactions

class data_rx : public action {
    public:
        my_test2(const scope& s) : action(this) {} 
        read_in_a rd_i {"rd_i"};
        q2m_xfer q2m {"q2m"};
        constraint c1 {...};
        bind b {this, rd_i.data, q2m.data};
    
    activity a {
        parallel {
            rd_i,
            q2m
        }
    } 
}

type_decl<data_rx> data_rx_decl;

Action class derives from library class

Activity defined as a tree of library class instances representing statements

Need to register the type in the PSS library
Different from ‘decltype’ in C++11
Activities

- Action "instances" can be thought of as "nodes" in an activity graph
- Optionally allows inline constraints
- Node may represent a variable randomization
- By default, activity statements execute sequentially
- \( \text{final}_n \) completes before \( \text{initial}_n \) starts

```markdown
action A {
    rand bit[3:0] f;
    ...
}

action top {
    A a1, a2;
    action int [0..5] i;
    activity {
        a1;
        i;
        a2 with { f < 10 };
    }
}
```
Activity – Robust Scheduling

- **Parallel branches start at the same time**
  - Initial action(s) in all branches have the same set of predecessors
  - No cross-branch dependencies

```plaintext
action top {
    A a1, a2;
    B b1, b2;

    activity {
        parallel {
            { a1, a2 };  
            { b1, b2 };  
        }
    }
}
```

- **Select statement randomly chooses one branch from the block**
  - Executes one and only one branch
  - Choice subject to other constraints

```plaintext
action top {
    A a; B b; C c;

    activity {
        a
        select {
            b;
            c;
        }
    }
}
```
Activity – Robust Scheduling

- If-Else block
- Do-while
- Repeat
- Foreach
  - For each element of an array
- Schedule
  - Actions in a schedule block execute in whatever order is legal given constraints
  - All actions must execute

```c
action top {
    A a;
    B b;
    C c;

    activity {
        a
        schedule {
            b;
            c;
        }
    }
}
```
Flow Objects: Dataflow & Scheduling

- **Flow objects are user-defined datatypes**
  - Special types of `struct`
  - May inherit from `struct` but not from each other
  - Action inputs or outputs

- **Buffer objects define sequential data/control flow**
  - Pre- or post-condition for action execution
  - Persistent storage; can be read after written

```plaintext
struct base_s {
    rand int size;
}

buffer data_buff_b : base_s {
    rand bit[31:0] start_addr;
    rand bit[1:0] mode;
}

stream data_strm_s : base_s {
    rand bit[1:0] inside [1..3] stop_bits;
    rand dir_enum direction;
}
```

- **Stream objects define parallel data/control flow**
  - Model parallel data flow
  - Message/notification exchange

- **State objects define the state of an element at a particular time**
  - State object writes must be sequential
  - Reads can be concurrent with other reads, but not writes
Buffer Object Semantics

- An action that inputs a buffer object must be bound to an action that outputs a buffer object of the same type
  - Outputting action can be referenced explicitly or implicitly
- Buffer object output can be connected to 0:N input actions
  - Must be of the same type
- Producing action must complete before execution of consuming action may begin

```vhdl
buffer data_buff_s {
  rand int[4..1024] size;
  rand bit[63:0] addr;
};

action cons_mem_a {
  input data_buff_s in_data;
};

action prod_mem_a {
  output data_buff_s out_data;
};
```
Stream Object Semantics

- An action that inputs a stream object must be bound to an action that outputs a stream object of the same type
  - Outputting action can be referenced explicitly or implicitly

- An action that outputs a stream object must be bound to an action that inputs a stream object of the same type
  - Inputting action can be referenced explicitly or implicitly

- Execution of outputting and inputting actions occur in parallel

```plaintext
stream data_strm_s {
  rand int[4..1024] size;
  rand bit[63:0] addr;
};

action cons_mem_a {
  input in_data;
};

action prod_mem_a {
  output out_data;
};
```
Defining Target-Specific Constraints

- Certain actions may require target resources
  - DMA Channel
  - Video pipe
  - Compress/Decompress engine
  - etc.

- Resources modeled as user-defined types
  - Specialized struct type

- Actions may claim a resource for the duration of their execution
  - Lock: excludes other actions from accessing the same resource
  - Share: no action may lock the resource until action completes

- Test defines how many resources are available
  - Pool defines how many are available
  - Pools may bind to actions
Resource Objects

- **Lock Example:**

  ```
  resource channel_s{...};
  pool [1] channel_s chans;
  action xfer_a {
    lock channel_s chan;
  };
  action xfer_b {
    lock channel_s chan;
  };
  ```

- **Share Example:**

  ```
  resource channel_s{...};
  pool [1] channel_s chans;
  action xfer_a {
    share channel_s chan;
  };
  action xfer_b {
    lock channel_s chan;
  };
  action xfer_c {
    share channel_s chan;
  };
  ```
Back to the Example: Resources

```
resource dma_channel_r {
    //implicit instance_id attribute
    ...
}

struct dma_xfer_params_s {
    rand bit[1:0] mode;
    rand bit[31:0] src_addr;
    rand bit[31:0] dst_addr;
    rand bit[5:0] channel;
}
```

```
action q2m_xfer {
    input data_stream_s src;
    output data_buff_b dst;

    lock dma_channel_r channel;
    rand dma_xfer_params_s params;

    constraint c1 { src.size == dst.size; }
    constraint params_c {
        params.mode == 'b01;
        params.src_addr == src.addr;
        params.dst_addr == dst.addr;
        params.channel == channel.instance_id;
    }
}
```

```
action base_xfer {
    lock dma_channel_r channel;
    rand dma_xfer_params_s params;
}
```

```
action q2m_xfer:base_xfer {
    input data_stream_s src;
    output data_buff_b dst;
    constraint c1 { src.size == dst.size; }
    constraint params_c {
        params.mode == 'b01;
        params.src_addr == src.addr;
        params.dst_addr == dst.addr;
        params.channel == channel.instance_id;
    }
}
```
Components & Pools

- Components are type namespaces
- Reusable groupings of
  - actions
  - pools
    - objects
    - resources
  - configuration parameters
- A pool is a collection of object/resource instances
  - *Bind* directive associates pools with actions
  - Specify which actions can exchange flow objects
  - Specify which actions contend for the same pool of resources
Components & Pools: C++

- Components are type namespaces

- Reusable groupings of
  - actions
  - pools
    - objects
    - resources
  - configuration parameters

- A pool is a collection of object/resource instances
  - **Bind** directive associates pools with actions
  - Specify which actions can exchange flow objects
  - Specify which actions contend for the same pool of resources

```cpp
class uart_c : public component {
    public:
        uart_c(const scope& s):component(this){}

    class uart_r : public resource {...};
    pool<uart_r> uart_p {"uart_p", 1};
    bind b1 {uart_p};

    class read_in_a : public action {
        public:
            read_in_a(const scope& s):action(this){}
            lock<uart_r> uart_l{"uart_l"};

            output<data_stream_s> out{"out");
            constraint c1 { ... };
    }
    type_decl<read_in_a> read_in;

    class write_out_a : public action {
        public:
            write_out_a(const scope& s):action(this){}
            input<data_stream_s> in{"in"};

            lock<uart_r> uart_l{"uart_l"};
    }
    type_decl<write_out_a> write_out;

};
```
Back to the Example

- The DMAC Component

```plaintext
component dmac_c {

    pool dma_channel_r chan_p;
    bind chan_p {*};

    action q2m_xfer_a {
        input data_stream_s in;
        output data_buff_b out;
        lock dma_channel_r chan;
    }

    action m2q_xfer_a {
        input data_buff_b in;
        output data_stream_s out;
        lock dma_channel_r chan;
    }

    action m2m_xfer_a {...}
}
```
Back to the Example

- The DMAC Component in C++

```cpp
class dmac_c : public component {
    public:
    dmac_c(const scope& s):component (this){}

    pool<dma_channel_r> chan_p;
    bind b1 {chan_p};

    class q2m_xfer_a : public action {
        public:
        q2m_xfer_a(const scope& s):action(this){}

        input<data_stream_s> in{"in"};
        output<data_buff_b> out{"out"};
        lock<dma_channel_r> chan{"chan"};
    };
    type_decl<q2m_xfer_a> q2m_xfer;

    class m2q_xfr_a : public action {...};
    class m2m_xfr_a : public action {...};
};
```
Back to the Example

- **Top-level Component**

```plaintext
component pss_top {

  uart_c uart0;
  dmac_c dma0;

  pool data_stream_s stream_p;
  bind stream_p {*};

  pool data_buff_b buff_p;
  bind buff_p {*};
}
```
Back to the Example

- **Top-level Component in C++**

```cpp
class top_c : public component {
public:
    top_c(const scope& s):component(this){}

    uart_c uart0 {"uart0"};
    dmac_c dma0 {"dma0"};

    pool<data_stream_s> stream_p {"stream_p"};
    bind b2 {stream_p};

    pool<data_buff_b> buff_p {"buff_p"};
    bind b2 {buff_p};
};
```
Creating a Test

- **Actions specify behaviors**
  - Actions define input/output to communicate with other actions
  - Actions claim resources that are target-specific

- **Activities define top-level scenarios**
  - Compound actions define high-level intent
  - Graphs define scheduling of actions

- **Resources & Flow Objects define additional scheduling constraints**
  - Locking resources prevents other actions from using them in parallel
  - Stream objects require another action to execute in parallel
  - Buffer objects allow another action to execute sequentially

- **Components group useful stuff for reuse**
Creating a Test: Loopback

- Receive data on UART and DMA into memory (in parallel)
  - `read_in_a` & `q2m_xfer` in parallel

- DMA from memory to UART and transmit (in parallel)
  - `m2q_xfer` & `write_out_a` action in parallel

```plaintext
action loopback_test {
    bind rd_i.data q2m.src;
    bind wr_o.data m2q.dst;
    bind q2m.dst m2q.src;

    activity {
        parallel {
            rd_i;
            q2m;
        }
        parallel {
            wr_o;
            m2q;
        }
    }
}
```
Creating a Test: Loopback

- Activity graph only needs to define critical intent [the "what"]
- Flow objects and resources constrain the possible scenarios
  - Tool can infer supporting actions [the "how"]

```
action loopback_test {
    bind wr_o.data m2q.dst;

    activity {
        rd_i;

        parallel {
            wr_o;
            m2q;
        }
    }
}
```
Creating a Test: Loopback

- Activity graph only needs to define critical intent [the "what"]
- Flow objects and resources constrain the possible scenarios
  - Tool can infer supporting actions [the "how"]

```plaintext
action loopback_test {
  activity {
    rd_i;
    wr_o;
  }
}
```
Creating a Test: Loopback

- Must make sure to prevent illegal inferencing
  - UART cannot read and write at the same time

```c
resource uart_r {...};
pool uart_r myuart;

action loopback_test {

  activity {
    rd_i;
    wr_o;
  }
}
```
Creating a Test: Loopback

- Can infer any actions that create a legal scenario
  - Subject to constraints
    - Object constraints
    - Resource constraints
    - Scheduling constraints

```plaintext
resource uart_r {...};
pool uart_r myuart;

action loopback_test {
  activity {
    wr_o;
  }
}
```
Creating a Test: Loopback

- Can infer any actions that create a legal scenario
  - Subject to constraints
    - Object constraints
    - Resource constraints
    - Scheduling constraints

```c
resource uart_r {...};
pool uart_r myuart;

action loopback_test {
  activity {
    wr_o;
  }
}
```
Creating a Test: Loopback in C++

class top_c : public component {
public:
  top_c(const scope& s): component(this) {}  

  uart_c uart0 {"uart0"};
  dmac_c dma0 {"dma0"};

  pool<data_stream_s> stream_p;
  bind b1 {stream_p}

  pool<data_buff_b> buff_p;
  bind b1 {buff_p}

class loopback_test_a : class action {
public:
  loopback_test_a(const scope& s): action(this) {}  
  action_handle<write_out_a> wr_o {"wr_o"};
  activity a {
    wr_o
  };
  type_decl<loopback_test_a> loopback_test;
};
Creating Portable Stimulus Models with the Upcoming Accellera Standard

Thank You!
Creating Portable Stimulus Models with the Upcoming Accellera Standard

Part 2

- Building System-Level Scenarios
- Generating Tests from Portable Stimulus
Sharon Rosenberg, Cadence Design Systems

BUILDING SYSTEM-LEVEL SCENARIOS
What are System-Level Scenarios?

- *The whole is greater than the sum of its parts!*
  - *And so are its bugs...*

- **Application use cases involve multiple IPs interoperating**
  - Example – read video off a mass-storage device, decode, split audio data from video frames, process by dedicated multi-media engines

- **Stress and performance use cases involve saturated utilization of shared resources**
  - Example – all processors and DMA-enabled controllers access a certain memory controller in parallel

- **System low-power use cases need to be crossed with functional scenarios**

- **System coherency of caches/TLBs requires coordinated pattern of accesses from CPUs and non-processor masters**
A Simple SoC Example

- Quad-core CPU that can write, read or copy memory buffers
- UART that can transmit or receive, but is not DMA-enabled
- Crypto engine that can encrypt and decrypt
- Multi channels system DMA
- Graphics SS that can decode images or videos
Modeling Targeted Behaviors

Step 1: Define component types and their operations as actions

Step 2: Compose actions into activities to specify scenarios and scenario building blocks

Step 3: Instantiate components and shared resources

Scenarios capture *pure intent*, abstracting from DUT configuration, verification environment, execution platform
Reuse IP Models

Actions are abstract, declarative, concise, well encapsulated units of behavior.

```
component crypto_c {
    action encrypt {
        input data_buff_s src_data;
        output data_buff_s dst_data;
        constraint {
            // operates on 128-byte blocks
            (src_buff.seg.size % 128) == 0;
            (dst_buff.seg.size % 128) == 0;

            // output is encrypted, input not
            !src_buff.encrypted;
            dst_buff.encrypted;
        }
    }
}
```

```
component uart_c {
    action write_out_a {
        input data_stream_s src_data;
        constraint src_data.direction == outwards;
    }

    action read_in_a {
        output data_stream_s dst_data;
        constraint dst_data.direction == inwards;
    }
}
```

```
component dma_c {
    resource struct channel_s {};
    pool [32] channel_s chan;
    bind chan *;

    action mem2mem_xfer_a {
        input data_buff_s src_data;
        output data_buff_s dst_data;
        lock channel_s channel;
        constraint src_data.seg.size == dst_data.seg.size;
    }
    ...
}
```
Processor cores are resources that can be locked or shared by other components' actions (e.g. for their control)

Attributes and constraints can be associated with resources
Overriding Types

- Override block may be specified in an action or a component

```plaintext
action reg2axi_top {
    override {
        type axi_write_action with axi_write_action_x;
        instance xlator.axi_action with axi_write_action_x2;
    }
    xlator_action xlator;
    ...
}

action reg2axi_top_x : reg2axi_top {
    override {
        type axi_write_action with axi_write_action_x4;
        instance xlator.axi_action with axi_write_action_x3;
    }
    ...
}
```

- Overrides are additive across extensions
- Overrides in a base type are replaced in the extension iff the type-instance is the same
Specifying Multi-IP Data Flows

```plaintext
action chain1 {
cpu_c::create_data_a write;
crypto_c::encrypt enc;
cpu_c::copy_data_a copy;
crypto_c::decrypt dec;
cpu_c::check_data_a check;
activity {
write;
enc;
copy;
dec;
check;
bind write.out_buff enc.src_buff;
bind enc.dst_buff copy.src_buff;
bind copy.dst_buff dec.src_buff;
bind dec.dst_buff check.in_buff;
};
}

action multi_chain {
activity {
schedule {
do chain1;
do chain2;
};
};
};
```

Resource conflict is automatically taken care of by serializing the contending actions.
Specifying Coordinated Flows

- A simple coherency scenario
  - CPU core writes data to cacheable region
  - A different core and a DMA read that same memory region

```
action simple_io_coherency {
  cpu_c::write_data write;
  cpu_c::copy_data_a copy;
  dma_c::mem2mem_xfer_a xfer;
  cpu_c::check_data_a check1, check2;

  activity {
    write with { out_buff.seg.cacheable == true; };
    parallel {
      { 
        copy with { core.instance_id != write.core.instance_id; };
        check1;
        bind copy.dst_buff check1.src_buff;
      }
      { 
        xfer;
        check2;
        bind xfer.dst_buff check1.src_buff;
      }
    }
    bind write.out_buff {copy.src_buff, dma_xfer.src_buff};
  }
};
```

Actions can be realized with true RTL CPUs / device, or with testbench bus transactors.
Layering System Power Concern

- Two power domains: A and B
  - Each power can be in mode S0 (active), S1 S2 (sleep modes)
- Subsystem operations depend on respective domain active state

Should reuse base model of behaviors and scenarios as is!
Defining Power Logic

```
enum power_state_e {S0, S1, S2};

state power_state_s {
    rand power_state_e dmn_A, dmn_B;

    constraint initial -> {
        dmn_A == S0 ;
        dmn_B == S0 ;
    }
};

component power_ctrl_c {
    pool power_state_s sys_pwr_statevar;

    action change_power_state {
        input power_state_s prev;
        output power_state_s next;
    }
};

extend pss_top {
    power_ctrl_c power_ctrl;
    bind power_ctrl.sys_pwr_statevar *;
};
```
Introducing Power Dependencies

```cpp
extend graphics_c::decode {
    input power_state_s curr_power_state;
    constraint curr_power_state.dmn_B == S0;
};

extend crypto_c::encrypt{
    input power_state_s curr_power_state;
    constraint curr_power_state.dmn_A == S0;
};
```

Dependencies layered on top of existing action definitions in a non-intrusive way.

```cpp
action encrypt_after_low_A {
    activity {
        do change_power_state with {
            next.dmn_A != S0;
        };
        do encrypt;
    };
}
```

Tool must infer additional action due to action precondition.

```cpp
change_pwr
```

Input state with a precondition

```cpp
pwr_state
dmn_A=S0
dmn_B=S0
```

```cpp
pwr_state
dmn_A=S0
dmn_B=S2
```

```cpp
pwr_state
dmn_A=S0
dmn_B=S0
```
Exercising Power Scenarios

A PSS compliance tool analyzes the entire scenario and trims the scenario space.

```cpp
action rand_traffic {
    activity {
        select {
            do cpu_c::copy;
            do crypto_c::encrypt;
            do graphics_c::decode;
        }
    }
}

action phased_pwr_traffic {
    activity {
        repeat (2) {
            do change_power_state;
            do rand_traffic;
        }
    }
}
```
Adnan Hamid, Breker Verification Systems

GENERATING TESTS FROM PORTABLE STIMULUS
PSS Test Generation Flow

PSS Model

PSS Tool

Verification Engine

CPU

DDR Controller

DMA

UART

Crypto

Graphics SS

SoC
Deployment Models

Interactive Test Generation
(Runtime Solving, Potentially limited portability)

Pre-Generated Test
(Generation-time Solving, Potentially limited reactivity)

On Target Generation
(Model + Tool running on SoC)
Exec Block Types

- Specify mapping of PSS entities to their implementation

```
#include <stdint.h>
void declared_func() {
    ...
}
void test_main() {
    do_run_start();
    fork_threads();
    do_run_end();
}
void thread0() {
    ...
    do_body();
    ...
};
void thread1() {
    ...
}
```

gcc -c test.c -DBARE_METAL

test.sh

Could be multiple threads on one core, or threads running on different cores
Using Code Templates in Exec-Body

- **Exec ‘body’ block** specifies implementation
  - Call `init_uart_rx`, specifying appropriate `stop_bits`
  - Call `gen_uart_traffic` with `stop_bits` and `size`

```sv
action read_in_a {
  output data_stream_s data;

  exec body SV = """
    init_uart_rx( {{data.stop_bits}});
    gen_uart_traffic( {{data.stop_bits}}, {{data.size}} );
  """
}
```

- **Exec 'declaration' block** can introduce declarations into generated test
  - UVM factory calls
  - Layered constraints

Can be arbitrary SV/UVM code
Platform 1: UVM Simulation

- Procedures implemented as SV tasks
  - Leverage platform infrastructure (VIP, registers)
- Test runs as a virtual sequence

```verilog
class uvm_simtest_base extends subsys_vseq;

  task init_uart_rx(byte unsigned stop_bits);
    m_uart_regs.LCR.STB = stop_bits;
    m_uart_regs.update();
  endtask

  task gen_uart_traffic(
    byte unsigned stop_bits,
    int sz);
    uart_vip_tx_seq tx_seq = new();
    assert(tx_seq.randomize() with {
      n_stop_bits == stop_bits;
      n_bytes == sz;
    });
    fork
      tx_seq.start(m_uart_vip.seqr);
    join_none
  endtask
endclass
```

Example Test

```verilog
class uvm_simtest1 extends uvm_simtest_base;

  virtual task body();
    //...

    // Action execution realization
    init_uart_rx(1);
    gen_uart_traffic(1, 128);
    //...

    // Action execution realization
    init_uart_rx(2);
    gen_uart_traffic(2, 27);

  endtask
endclass
```
Using Code Templates in Exec-Body

- Exec ‘body’ block specifies implementation
  - Call init_uart_rx, specifying appropriate stop_bits
  - Call gen_uart_traffic with stop_bits and size

```
action read_in_a {
  output data_stream_s data;

  exec body C = ""
    init_uart_rx({{data.stop_bits}});
    gen_uart_traffic({{data.stop_bits}}, {{data.size}});
  ""
}
```

- Exec 'declaration' block can introduce top-level declarations into generated C test
  - Types
  - Global objects
Platform 2: Software Driven Emulation

- Procedures implemented as C functions
  - Write directly to UART registers
  - Trigger UART traffic by writing to the UART VIP’s snoop address

```c
extern void write32(uint32_t *addr, uint32_t data);
extern uint32_t read32(uint32_t *addr);
extern uint32_t *UART_BASE;
extern uint32_t *UART_VIP_SNOOP_ADDR;
#define UART_LCR_OFFSET 3
#define UART_LCR_STB 2

void init_uart_rx(uint8_t stop_bits) {
    uint32_t lcr = read32(&UART_BASE[UART_LCR_OFFSET]);
    lcr &= ~(1 << UART_LCR_STB));
    lcr |= (stop_bits << UART_LCR_STB);
    write32(&UART_BASE[UART_LCR_OFFSET], lcr);
}

void gen_uart_traffic(uint8_t stop_bits, int sz) {
    // Write to the UART VIP snoop address
    // to trigger sending traffic.
    write32(UART_VIP_SNOOP_ADDR,
            (sz & 0xFFFF) | (stop_bits << 16));
}
```

Generated by "exec body C" template

```c
int main(int argc, char **argv) {
    //...
    init_uart_rx(1);
    gen_uart_traffic(1, 128);
    init_uart_rx(2);
    gen_uart_traffic(2, 27);
    return 0;
}
```

API Implementation

Example Test
Platform 3: Post-Si Host Bus Adapter

- Procedures implemented as C functions
  - Send PCIe TLPs to access UART
  - Send serial traffic via host UART

```c
void init_uart_rx(uint8_t stop_bits) {
    uint32_t lcr = pcie_read32(&UART_BASE[UART_LCR_OFFSET]);
    lcr &= ~(1 << UART_LCR_STB));
    lcr |= (stop_bits << UART_LCR_STB);
    pcie_write32(&UART_BASE[UART_LCR_OFFSET], lcr);
}

void gen_uart_traffic(uint8_t stop_bits, int sz) {
    int i;
    struct termios opt;

    // Create random data
    uint8_t *data = (uint8_t *)malloc(sz);
    for (i=0; i<sz; i++) { data[i] = rand(); }

    // Configure the stop bits
    tcgetattr(UART_FD, &opt);
    opt.c_cflag &= (~CSTOPB);
    opt.c_cflag |= (sz==2)?CSTOPB:0;
    tcsetattr(UART_FD, &opt);

    // Send data
    write(UART_FD, data, sz);
    free(data);
}
```
Using Import Functions in Exec-Body

- External procedures implement the test
  - Program UART receive mode
  - Trigger generation of UART traffic

```c
// Initializes the UART to receive
import void init_uart_rx(bit[1:0] stop_bits);

// Triggers an external agent to generate UART traffic
import void gen_uart_traffic(bit[1:0] stop_bits, int sz);
```

- Exec ‘body’ block specifies implementation
  - Call init_uart_rx, specifying appropriate stop_bits
  - Call gen_uart_traffic with stop_bits and size

```c
action read_in_a {
    output data_stream_s data;
    exec body {
        init_uart_rx(data.stop_bits);
        gen_uart_traffic(data.stop_bits, data.size);
    }
}
```
Platform 1: UVM Simulation

- Procedures implemented as SV tasks
  - Leverage platform infrastructure (VIP, registers)
- Test runs as a virtual sequence

```h sữam
class uvm_simtest1 extends uvm_simtest_base;

virtual task body();
//...

// Action execution realization
init_uart_rx(1);
gen_uart_traffic(1, 128);
//...

// Action execution realization
init_uart_rx(2);
gen_uart_traffic(2, 27);
endtask
endclass
```

```
Platform 1: UVM Simulation

- Procedures implemented as SV tasks
  - Leverage platform infrastructure (VIP, registers)
- Test runs as a virtual sequence

```h sữam
class uvm_simtest_base extends subsys_vseq;

    task init_uart_rx(byte unsigned stop_bits);
        m_uart_regs.LCR.STB = stop_bits;
        m_uart_regs.update();
    endtask

    task gen_uart_traffic(
        byte unsigned stop_bits,
        int sz);
        uart_vip_tx_seq tx_seq = new();
        assert(tx_seq.randomize() with {
            n_stop_bits == stop_bits;
            n_bytes == sz;
        });
        fork
            tx_seq.start(m_uart_vip.seqr);
        join_none
    endtask
endclass
```
Platform 2: Software Driven Emulation

- Procedures implemented as C functions
  - Write directly to UART registers
  - Trigger UART traffic by writing to the UART VIP’s snoop address

```c
extern void write32(uint32_t *addr, uint32_t data);
extern uint32_t read32(uint32_t *addr);

extern uint32_t *UART_BASE;
extern uint32_t *UART_VIP_SNOOP_ADDR;

#define UART_LCR_OFFSET 3
#define UART_LCR_STB 2

void init_uart_rx(uint8_t stop_bits) {
    uint32_t lcr = read32(&UART_BASE[UART_LCR_OFFSET]);
    lcr &= (~((1 << UART_LCR_STB)));  
    lcr |= (stop_bits << UART_LCR_STB);
    write32(&UART_BASE[UART_LCR_OFFSET], lcr);
}

void gen_uart_traffic(uint8_t stop_bits, int sz) {
    // Write to the UART VIP snoop address
    // to trigger sending traffic.
    write32(UART_VIP_SNOOP_ADDR, (sz & 0xFFFF) | (stop_bits << 16));
}
```

Example Test

```c
int main(int argc, char **argv) {
    //...

    // Action execution realization
    init_uart_rx(1);
    gen_uart_traffic(1, 128);

    // Action execution realization
    init_uart_rx(2);
    gen_uart_traffic(2, 27);

    return 0;
}
```
Platform 3: Post-Si Host Bus Adapter

- Procedures implemented as C functions
  - Send PCIe TLPs to access UART
  - Send serial traffic via host UART

```c
void init_uart_rx(uint8_t stop_bits) {
    uint32_t lcr = pcie_read32(&UART_BASE[UART_LCR_OFFSET]);
    lcr &= ~(1 << UART_LCR_STB));
    lcr |= (stop_bits << UART_LCR_STB);
    pcie_write32(&UART_BASE[UART_LCR_OFFSET], lcr);
}

void gen_uart_traffic(uint8_t stop_bits, int sz) {
    int i;
    struct termios opt;

    // Create random data
    uint8_t *data = (uint8_t *)malloc(sz);
    for (i=0; i<sz; i++) { data[i] = rand(); }

    // Configure the stop bits
    tcgetattr(UART_FD, &opt);
    opt.c_cflag &= (~CSTOPB);
    opt.c_cflag |= (sz==2)?CSTOPB:0;
    tcsetattr(UART_FD, &opt);

    // Send data
    write(UART_FD, data, sz);
    free(data);
}
```
Using HSI Abstraction in Exec-Body

class uart_c : public component {
public:
    uart_c(const scope& s):component(this){}

class read_in_a : public action {
public:
    read_in_a(const scope& s):action(this){}

    output<data_stream_s> out{"out"};
    constraint c1 { ... };
    constraint c2 { ... };

    uart_hsi hsi{"hsi"};
    randv<bit<0,0>> stop_bits{"stop_bits"};

    void body() {
        hsi.init_uart_rx(stop_bits);

        // drive input data on VIP
    }

    type_decl<read_in_a> read_in;
};
class uart_c : public component {
    public:
    uart_c(const scope& s):component(this){}

class read_in_a : public action {
    public:
    read_in_a(const scope& s):action(this){}

    output<data_stream_s> out{"out"};
    constraint c1 { ... };
    constraint c2 { ... };

    uart_hsi hsi{"hsi"};
    randv<bit<0,0>> stop_bits{"stop_bits"};

    void body() {
        hsi.init_uart_rx(stop_bits);

        // drive input data on VIP
    }
};

type_decl<read_in_a> read_in;
class uart_c : public component {
public:
    uart_c(const scope& s):component(s) {

class read_in_a : public action {
public:
    read_in_a(const scope& s):action(this) {

output<data_stream_s> out{"out"};
    constraint c1 { ... };
    constraint c2 { ... };

    uart_hsi hsi{"hsi"};
    randv<bit<0,0>> stop_bits{"stop_bits"};

    void body() {
        hsi.init_uart_rx(stop_bits);

        // drive input data on VIP
    }
};
type_decl<read_in_a> read_in;
};
Platform 3: Post-Si Host Bus Adapter

```cpp
class uart_c : public component {
    public:
    uart_c(const scope& s):component(this){}

class read_in_a : public action {
    public:
    read_in_a(const scope& s):action(this){}
    output<data_stream_s> out{"out"};
    constraint c1 {...};
    constraint c2 {...};

    uart_hsi hsi{"hsi"};
    randv<bit<0,0>> stop_bits{"stop_bits"};

    void body() {
        hsi.init_uart_rx(stop_bits);
        // drive input data on VIP
    }

    type_decl<read_in_a> read_in;
};
```
Creating Portable Stimulus Models with the Upcoming Accellera Standard

Thank You!
Creating Portable Stimulus Models with the Upcoming Accellera Standard

Part 3

- Coverage in Portable Stimulus
- The Hardware/Software Interface Library
- Conclusion
Srivatsa Vasudevan, Synopsys

COVERAGE IN PORTABLE STIMULUS
Demystifying Coverage

- What coverage is and is NOT in Portable Stimulus
- Defining scenario coverage
- Coverage monitoring
- Usage examples
What is Portable Stimulus Coverage?

- **Code Coverage?** No
- **Functional Coverage?** Closer
  - Covergroups? Could be, but not at implementation/protocol level.
- **Test Coverage?** Ok, but can’t we do better?
Portable Stimulus Coverage
Opportunity & Challenge

- Examples of system level coverage:
  - Connectivity and addressability testing
  - Power state sequencing
  - Resource utilization - Did all internal memories get used by DMA tests?

- Introduction of random => Need coverage to confirm usefulness

- Portability challenge – collecting coverage in non-simulation environments
  - Lack of visibility in HW-based platforms makes traditional coverage collection difficult
Reimagined Coverage

PS Enabled Development Framework

- Architect
- Use Cases, PSS Models
- IP Design/DV
  - IP DUT, PSS Models
- System Design/DV
  - SUT, PSS Models
- Apps Eval
  - Scenarios
- Software
  - Device Drivers
- Test
  - Scenarios

- TLM Model
  - SystemC
- Simulation
  - SV, UVM
  - C/C++
- Evaluation, Customer Boards
  - C/C++
- Virtual, Emulator, Silicon
  - C/C++
- Tester
  - Multiple

- C/C++
Types of Coverage in Portable Stimulus

- **Action Coverage**
  - Were all (or a specified subset of) defined actions executed?

- **Scenario (Action Sequence) Coverage**
  - What legal sequences of actions were exercised? Aka “control path coverage”

- **Datapath Coverage**
  - Were all legal sources and sinks for an action sequence datapath (input/output) covered?

- **Value Coverage**
  - Think covergroups for attributes (config values, state values, … )

- **Resource Coverage**
  - Any resources added to a resource pool that went unused?

- **Crosses of any of the above types**
Defining Scenario (Action Sequence) Coverage

- Scenarios are all legal behavior defined between entry and exit points
  - Choices are made by the tool between these points
    - e.g. alternative actions, resource usage, data source

- If we can enumerate the choices, we can measure coverage of them
  - In theory a tool could also target this coverage
    - i.e. make choices based on what has/hasn’t been covered

- Warning: with great power comes great responsibility
  - Be careful of the number of choices between your entry and exit points
  - Don’t try to target a coverage with more choices than atoms in the universe
Monitoring Coverage

- **Stimulus monitoring**
  - Generation time tool can output what it generated/scheduled
  - As long as test “passes,” the coverage data is valid

- **Runtime State monitoring**
  - Requires generation of monitoring code
  - May be C/C++ code running on target CPU
    - e.g. data sent out “trickbox” mechanism
  - May be “off-chip” monitoring via test ports or other communication ports
Usage Examples

- Cover – Resource utilization
  - cover resource mem with (type == SRAM)

- Cover – Uart example

- Cover – DMA example
Simple Example: UART

```
stream data_stream_s {
    rand int size;
    rand dir_enum direction;
    rand bit[1:0] inside [1..3] stop_bits;
}
buffer data_buff_b {
    rand int size;
}
```

```
action q2m_xfer {
    input data_stream_s src;
    output data_buff_b dst;
}
```

```
action read_in_a {
    output data_stream_s data;
}
```

```
action write_out_a {
    input data_stream_s data;
    coverspec {
        size_cp : coverpoint data.size {
            bins size_bins [1..20]:1;
        }
    }
}
```

```
DMA pkt stream into mem buffer
```

```
DMA mem buffer into pkt stream
```

```
Generate pkt stream
```

```
UART
```

```
MEM
```

```
DMAC
```

```
Command
```

```
Data
```
Command

abstract action move_data_a {
    input data_buff_s src_buff;
    output data_buff_s dst_buff;
    constraint {src_buff.seg.size == dst_buff.seg.size};

coverspec {
    src_cp : coverpoint src_buff.location;
    dst_cp : coverpoint dst_buff.location;
    srcXdst : cross src_cp, dst_cp;
    size_cp : coverpoint src_buff.seq.size {
        bins size_bins = [1..20]:1;
    }
}
}
abstract action move_data_a {
    input data_buff_s src_buff;
    output data_buff_s dst_buff;
    constraint {src_buff.seg.size == dst_buff.seg.size};

    coverspec {
        constraint {src_buff.seg.size != 10};
        src_cp : coverpoint src_buff.location;
        dst_cp : coverpoint dst_buff.location;
        srcXdst : cross src_cp, dst_cp;
        size_cp : coverpoint src_buff.seq.size {
            bins size_bins = [1..20]:1;
        }
    }
}
Karthick Gururaj, Vayavya Labs
Sandeep Pendharkar, Vayavya Labs

THE HARDWARE/SOFTWARE INTERFACE LIBRARY
The Story so far...

- Importance of Portability of test cases
  - To different environments
  - And different platforms
- Capturing complex use cases
- Measuring Coverage

Is that all there is to it?
Need for HW-SW Interface in PS

Hardware-Software Interface spec is required for “real portability” across environments
What is HSI?

- **Hardware/Software Interface layer is**
  - …an abstraction responsible for device management
    - Device initialization, operations such as configure, transmit/receive
    - Registration of device capabilities
  - …set of constructs for capturing the Hardware aspects required to implement the abstraction
    - Programming registers, setting up descriptor chains, interrupt properties and handling, …
    - Capture all programming sequences
  - …to summarize: construct the programmer’s view of a device agnostic to the underlying verification environment
Scenarios and HW/SW Interface

- Captures the test intent
- Uses Driver APIs specified in the Driver API Interface Layer
- Device agnostic, but specific to a device-function (category)
- Is device specific
- Sequences for configuration, initialization, descriptor management, data transfer, …
- Interrupt handling
- Publish device capabilities
What HSI Enables

Ensures Portability of Scenarios across Environments
What HSI Enables

- UART A
- MEM
- DMAC A

Driver for DMAC-A
Driver for UART-A

HSI Spec for DMAC-A
HSI Spec for UART-A

Test scenarios

- UART B
- MEM
- DMAC B

Driver for DMAC-B
Driver for UART-B

HSI Spec for DMAC-B
HSI Spec for UART-B

Enables Portability of Scenarios across Devices/SoCs
HW/SW Interface Spec Elements

- Registers
- FIFOs
- Virtual registers
- Descriptor management
- Interrupt management
- Sequences
- Device capabilities
DMA Allocation Revisited

resource pool rp with 4 instances of type channel_s

action type mem2mem_xfer_a {
    input data_buff_s of type src_data;
    output data_buff_s of type dst_data;
    resource claim rc of type channel_s;
    constraint src_data.seg.size == dst_data.seg.size;
};

action type mem2queue_xfer_a {
    input data_buff_s of type src_data;
    output data_stream_s of type dst_data;
    resource claim rc of type channel_s;
    constraint src_data.seg.size == dst_data.size;
};

action type queue2mem_xfer_a {
    input data_stream_s of type src_data;
    output data_buff_s of type dst_data;
    resource claim rc of type channel_s;
    constraint src_data.size == dst_data.seg.size;
    constraint src_data.direction == inwards;
};

Declare Reg
    as a register bank
    With Channel as array of register bank
    With Register src of 32b;
    With Register dst of 32b;
    With Register ctrl of 32b
    With field src_incr;
    ...

Declare Intr
    as interrupt line
    With xfer_end as array of interrupts
    With Status
    Reg.Intr_STS[rc].xfer_end;
    Enable by Reg.Intr[rc].xfer_end=1;
    Disable by Reg.Intr[rc].xfer_end=0;
    ...

Note: Example in pseudocode, not actual representation
#include "pss.h"

class dma_src : public pss::reg
{
    public:
    dma_src(/* ... */) : pss::reg {
        description("Source address")
        , offset(0x0)
        , width(32)
        , access(pss::PSS_ACCESS_RW)
        , reset(0x0)
    }
};

class dma_dst : public pss::reg
{
    public:
    dma_dst(/* ... */) : pss::reg {
        description("Destination address")
        , offset(0x4)
        , width(32)
        , access(pss::PSS_ACCESS_RW)
        , reset(0x0)
    }
};

class channel_regs : public pss::reg_group
{
    public:
    dma_src src{"src"};
    dma_dst dst{"dst"};
    /* Other registers */
};

class dmac_regs : public pss::reg_group
{
    public:
    pss::vector<channel_regs> channel{"channel", 8};
    /* Other registers */
};

class dmac_interrupts : public pss::intr_line
{
    public:
    pss::intr_event xfer_done{"xfer_done"};
    /* ... */
};

class dmac : public pss::hsi
{
    public:
    dmac(/* ... */) { }
    void build(void);
    void mem2mem_xfer(void);
    void mem2queue_xfer(void);
    void queue2mem_xfer(void);

    dmac_regs regs;
    dmac_interrupts intr;
};
DMAC HSI Specification

```cpp
void dmac::build(void)
{
    intr.xfer_done
        .event_type(pss::PSS_STATUS)
        .enable(PSS_ANON_FUNC({regs.intr_enable.xfer_done = 1;}))
        .disable(PSS_ANON_FUNC({regs.intr_enable.xfer_done = 0;}))
        .get_status(PSS_EXPR({regs.intr_status.xfer_done == 1;}))
}

void dmac::mem2mem_xfer(dma_xfer_request &req)
{
    regs.channel[req.rc].src = req.src_data.address();
    regs.channel[req.rc].dst = req.dst_data.address();
    regs.channel[req.rc].size = req.src_data.size();
    regs.channel[req.rc].ctrl.src_incr = FIXED;
    regs.channel[req.rc].ctrl.dst_incr = INCR;
    regs.intr_enable.xfer_done = 1;
    regs.channel[req.rc].enable = 1;
    wait(intr.xfer_done);
}
```
Truly Portable Stimulus

Capabilities  Init/configure  Transmit/Receive

Sequences

Interrupt management  Descriptor Management

Register Access

Registers  FIFOs  Virtual Registers

Test bench
Bare metal drivers for bring up/validation

Firmware/OS Device Drivers
CONCLUSION

Faris Khundakjie, Intel
Portable stimulus is a perfect solution for many real problems we have today – even within a single platform.

Portable stimulus can stretch productivity and quality across platforms, users, integrations, and configurations.

Portable Stimulus Standard is a serious and timely industry effort under Accellera.

How this standard offers unique concepts and constructs (components, actions, flow objects and resources) to build powerful scenarios that map with flexibility to target platforms.
We Hope You Will…

Participate in shaping this promising standard with your suggestions, use cases and requirements through:

- Your company’s Accellera representation
- EDA vendor voicing your thoughts
- Contacting any of the speakers or PSWG officers

Be an agent of change

- Rethink verification and validation efficiency for your team and consumers
- Cross the aisle and communicate with peers in other platforms to accomplish more reuse with portable stimulus
We Thank…

Accellera and DVCon 2017 for offering PSWG the opportunity and real estate to deliver this tutorial to the community

All speakers who spent several hours and weeks preparing and improving this tutorial

All PSWG members for their feedback to improve tutorial’s message and content
Thank You!!