An Introduction to the Accellera Portable Stimulus Standard
Today's Speakers

• Introduction: Sharon Rosenberg, Sr. Solution Architect, Cadence
• Portable Stimulus Concepts & Constructs: Tom Fitzpatrick, Strategic Verification Architect, Mentor, a Siemens Business
• Building System-Level Scenarios: Sharon Rosenberg, Sr. Solution Architect, Cadence
• Generating Tests from Portable Stimulus: David Kelf, Advisor, Breker Verification Systems
• Hardware/Software Interface Library: Karthick Gururaj, Principal Architect, Vayavya Labs
• Coverage in Portable Stimulus: Piyush Sukhija, Sr. Verification Group Staff Engineer, Synopsys
• Conclusion: David Kelf, Advisor, Breker Verification Systems
PORTABLE STIMULUS
THE NEXT LEAP IN VERIFICATION & VALIDATION PRODUCTIVITY

Sharon Rosenberg, Cadence
A Brief History of Verification

Verification Productivity

Directed Testing

ad hoc Verification teams

System Verilog
CRV & MDV

Standard Methodology
UVM
VMM
OVM

Portable Stimulus

Verification?
SoC-level Verification

• Design complexity continues to increase
• Outstripping Verification Productivity (again)
• State space at system-level is too big for effective UVM Constrained-Random
• Multiple platforms
• Need to reuse Test Intent
  – Higher abstraction
  – Reuse from block to system
  – Map to different platforms
Reuse of Test Intent Across Platforms/Users

- Different tests used throughout a project
  - Wastes Time
  - Error Prone
- UVM constrained-random is too limited for directing SoC-level testing
- C tests usually directed for specific use-cases
  - Hard to create
  - Miss corner cases
Reuse of Test Intent Across Platforms/Users

• Single specification of test intent is critical
• Constrain and randomize at the Scenario Level by capturing:
  – interactions
  – dependencies
  – resource contention
• Abstraction lets tool automate generation
  – of Correct-by-construction legal tests
  – for Multiple targets
  – with Target-specific customization
PSS Tool Operating in a UVM Environment

PSS tool composes UVM sequences for existing UVM environment from abstract use case specification.

Tool handles verification checks and coverage collation against original spec.

Streamlines testbench authoring using an “executable specification”
PSS Tool Operating in a SoC Environment

PSS tool generates synchronized IO data and SW tests

Use cases may be reused from other process stages including simulation

A single scenario can drive complete test sets for emulation, reusing VIP
What Portable Stimulus Is Not

• A UVM replacement
• A reference implementation
• One forced level of abstraction
  – Expressing intent from different perspectives is a primary goal
• Monolithic
  – Representations would typically be composed of portable parts
• Two standards
  – DSL and C++ input formats describe 1:1 semantics
  – Tools shall consume both formats
• Just stimulus
  – Models Verification Intent
  – Stimulus, checks, coverage, scenario-level constraints
Tom Fitzpatrick, Mentor, *a Siemens Business*

CONCEPTS & CONSTRUCTS
The Steps to Creating a Portable Stimulus Model

**Step 1:** Identify Behaviors

**Step 2:** Identify Data Flow

**Step 3:** Where do Actions Execute?

**Step 4:** What System Resources are required?

component uart_c {
  resource uart_r {...};
  pool [1] uart_r uart_p;
  bind uart_p {*};

  action read_in_a {
    output data_str data;
    lock uart_r u_busy;
  }

  action write_out_a {
    input data_str data;
    lock uart_r u_busy;
  }
}
The Steps to Creating a Portable Stimulus Model

• Step 1: Identify Behaviors
• Step 2: Identify Data Flow
• Step 3: Where do Actions Execute?
• Step 4: What System Resources are required?

component dma_c {
    resource chan_r {...};
    pool [1] chan_r chan_p;
    bind chan_p {*};
}

action u2m_a {
    input data_str src_data;
    output mem_b dst_data;
    lock chan_r chan;
}

action m2u_a {
    input mem_b src_data;
    output data_str dst_data;
    lock chan_r chan;
}

action m2m_a {...}

component

© Accellera Systems Initiative

© Accellera Systems Initiative
Defining the Data Structures

- Portable Stimulus only cares about the structure of data
- Data type defines scheduling constraints
  - buffer = sequential
  - stream = parallel

```c
struct data_seg_s {
    rand int[4..1024] size;
    rand bit[63:0] addr;
};

buffer mem_b : data_seg_s {
};

enum data_dir_e {in, out};

stream data_str : data_seg_s {
    rand data_dir_e dir;
    rand int[1..2] stop_bits;
};
```
Data Flow Objects

• Specialized struct objects
  – Each may inherit from a struct; May not inherit from each other

```c
buffer
stream
state

[rand] struct_fields;
```

observed behavior

Stream = Parallel
Buffer = Sequential

© Accellera Systems Initiative 15 10/2/17
Adding Data Constraints

- Data constraints limit the solution space
- Actions that share a data object must have compatible constraints

```
action read_in_a {
    output data_str data;
    lock uart_r u_busy;
    constraint data.dir == in;
};

action u2m_a {
    input data_str src_data;
    output mem_b dst_data;
    lock chan_r chan;
    constraint src_data.dir == in;
    constraint src_data.size == dst_data.size;
};

action m2m_a {
    input mem_b src_data;
    output mem_b dst_data;
    lock chan_r chan;
    constraint src_data.size == dst_data.size;
    constraint src_data.addr != dst_data.addr;
};
```
buffer mbuf {
    bit[3:0] x;
    constraint {x < 10;}
};

action do_b {
    input mbuf m;
    constraint {m.x < 12;}
};

action do_a {
    output mbuf m;
    constraint {m.x > 2;}
};

action do_g {
    output mbuf m;
    constraint {m.x > 8;}
};

action do_p {
    input mbuf m;
    constraint {m.x < 8;}
};
Assembling the Top-Level Model

```component pss_top {
    uart_c uart0;
    dma_c dma0;

    pool data_str stream_p;
    bind stream_p {*};
    pool mem_b mem_p;
    bind mem_p {*};
}
```
Assembling the Top-Level Model

```component pss_top {
  uart_c uart0;
  dma_c dma0;

  pool data_str stream_p;
  bind stream_p {*};
  pool mem_b mem_p;
  bind mem_p {*};

  action loopback_test {
    uart_c::read_in_a rd_i; dma_c::u2m_a u2m;
    uart_c::write_out_a wr_o; dma_c::m2u_a m2u;
    bind rd_i.data u2m.src_data;
    bind wr_o.data m2u.dst_data;
    bind u2m.dst_data m2u.src_data;

    activity {
      parallel { rd_i; u2m; }
      parallel { wr_o; m2u; }
    }
  }
}
activity { that; do an_a; parallel {a1, a2}; sequence {a3, a4}; select {a5, a6}; schedule {a7, a8}; if (i == 0) {a9;} else {a10;} repeat (2) {a11, a12}; foreach (arr[j]) { a13 with {a13.val == arr[j];}; }

Action instance traversal
Anonymous action traversal
Subject to flow/resource constraints
Assembling the Top-Level Model

```plaintext
component pss_top {
  uart_c uart0;
  dma_c dma0;

  pool data_str stream_p;
  bind stream_p {*};
  pool mem_b mem_p;
  bind mem_p {*};

  action loopback_test {
    uart_c::read_in_a rd_i; dma_c::u2m_a u2m;
    uart_c::write_out_a wr_o; dma_c::m2u_a m2u;
    bind rd_i.data u2m.src_data;
    bind wr_o.data m2u.dst_data;
    bind u2m.dst_data m2u.src_data;

    activity {
      parallel { rd_i; u2m; }
      parallel { wr_o; m2u; }
    }
  }
}
```
Assembling the Top-Level Model

```
component pss_top {
    uart_c uart0;
    dma_c dma0;
    pool data_str stream_p;
    bind stream_p {*};
    pool mem_b mem_p;
    bind mem_p {*};

    action loopback_test {
        uart_c::read_in_a rd_i;
        uart_c::write_out_a wr_o;

        activity {
            rd_i;
            wr_o;
        }
    }
}
```
Assembling the Top-Level Model

component pss_top {
    uart_c uart0;
    dma_c dma0;

    pool data_str stream_p;
    bind stream_p {*};
    pool mem_b mem_p;
    bind mem_p {*};

    action loopback_test {
        uart_c::read_in_a rd_i;
        uart_c::write_out_a wr_o;

        activity {
            rd_i;
            wr_o;
        }
    }
}
Mapping to Implementation

• exec blocks define target language implementation for actions
  – Target-language templates (C or SV)
  – Import external procedures from legacy code

• Exec blocks can only be defined for leaf-level actions
  – activity and exec body block are mutually exclusive

import void init_uart_rx(bit[1:0] stop_bits);
import void gen_uart_traffic(bit[1:0] stop_bits, int sz);
action read_in_a {
  output data_str data;
  lock uart_r u_busy;
  constraint data.dir == in;

  exec body {
    init_uart_rx( data.stop_bits );
    gen_uart_traffic( data.stop_bits , data.size );
  }
}

"Moustache" notation to refer to PSS fields

class uvm_test1 extends uvm_test_base;

  virtual task body();
    // ...
    init_uart_rx(1);
    gen_uart_traffic(1, 128);
    // ...
    init_uart_rx(2);
    gen_uart_traffic(2, 27);
  endtask
endclass
component uart_c {
  resource uart_r 
  pool [1] uart_r uart_p;
  bind uart_p (*)
}

action read_in_a {
  output data_str data;
  lock uart_r u_busy;
  constraint data.dir == in;
  exec body ...}

action write_out_a { ...}
}

component dma_c {
  resource chan_r 
  pool [1] chan_r chan_p;
  bind chan_p (*)
}

action u2m_a {
  input data_str src_data;
  output mem_b dst_data;
  lock chan_r chan;
  constraint src_data.dir == in;
  constraint src_data.size == dst_data.size;
  exec body ...}

action m2u_a { ...}

action m2m_a { ...}
}

component pss_top {
  uart_c uart0;
  dma_c dma0;
  pool data_str stream_p;
  bind stream_p (*);
  pool mem_b mem_p;
  bind mem_p (*);
  action loopback_test {
    uart_c::read_in_a rd_i;
    activity {
      rd_i;
      wr_o;
    }
  }
}

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;
}
BUILDING SYSTEM-LEVEL SCENARIOS
A Sub-system Example
Sub-system UVM View

virtual Sequences → BUS VIP

UART VIP → virtual Sequences

Command

Data

UART → DMAC

DMAC → DDR MEM

© Accellera Systems Initiative 2017

10/2/17
A Simple SoC Example

Virtual sequence
Cannot be reused

Quad-core CPU:
Write, Read or Copy
memory buffers

CPU

Crypto

Encrypt or
Decrypt Data

Decode Images
or Videos

DMAC

DDR MEM

Graphics

Multi-channel System DMA

Transmit or
Receive Data

UART VIP

UART

UART In fast platforms mostly not usable

Virtual Sequences

© Accellera Systems Initiative
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

So How do We Achieve SoC Scenarios and Enable Reuse?
A Simple SoC Example

Step 1: Define component types and their operations as actions with composition rules

Step 2: Compose actions into activities to specify scenarios

- **UART**
  - read
  - write
  - transfer
  - read_in
  - write_out

- **DMAC**
  - transfer

- **Crypto**
  - encrypt
  - decrypt

- **CPU**
  - write
  - copy
  - read

- **DDR MEM**
  - decode
Actions are abstract, declarative, concise, well encapsulated units of behavior.

One action should not assume the existing of another action.
SW Operations Modeling

Step #1: Modeling

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.

```plaintext
component cpu_c {
    abstract action sw_operation {
        lock core_s core;
    };

    action check_data : sw_operation {
        input data_buff_s src_data;
    };

    action write_data : sw_operation {
        output data_buff_s dst_data;
    };

    action copy_data : sw_operation {
        input data_buff_s src_data;
        output data_buff_s dst_data;
        constraint c1 {src_data.size == dst_data.size;}
    };
}

resource struct core_s {
    rand core_tag_e core_tag;
    rand cluster_tag_e cluster_tag;
    constraint {
        cluster_tag == CLUSTER_A -> core_tag inside [CORE_A0, CORE_A1];
        cluster_tag == CLUSTER_B -> core_tag inside [CORE_B0, CORE_B1];
    }
}

component pss_top {

    pool [4] core_s chan;
    bind core_s *;
}
```
Specifying Multi-IP Data Flows
Step #2: Scenario Creation

```
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 {
    do write;
    do enc;
    do copy;
    do dec;
    do 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
Step #2: Scenario Creation

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

```c
action simple_io_coherency {
    cpu_c::write_data write;
    cpu_c::copy_data_a copy;
    dma_c::mem2mem_xfer_a xfer;

    activity {
        write with { out_buff.seg.cacheable == true; };
        repeat (20) {
            parallel {
                copy with { core.instance_id != write.core.instance_id; }
                xfer;
            }
        }
        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

**Step #1: Modeling**

```plaintext
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 *;
}
```

- **State object representing aggregate system power state**
- **Enum attribute for each domain**
- **Both domains start out active**
- **Power transition action reads the previous power state and establishes a new state**
- **State variable is bound to actions' inputs/outputs by state type**
Introducing Power Dependencies

```java
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;
};
```

Input state with a precondition

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

```java
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

PSS also supports overrides of types and instances
Exercising Power Scenarios  
Step #2: Scenario Creation

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

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

```cpp
action phased_pwr_traffic {  
    activity {  
        repeat (2) {  
            do change_power_state;  
            do rand_traffic;  
        };  
    };  
};
```

This is where PSS takes solving to the next level.
Building system-level scenarios summary

- PSS DNA is built for portability and reuse
  - Applicable both for bottom-up and top-down reuse
  - Can reuse both the model and scenarios intent

- PSS Solving capabilities allows automation of system concerns
  - Takes randomization and constraint-solving to the next level:
    Timing, resources, configuration, synchronization, ...

- We gathered a set of usage examples to demonstrate these capabilities
  - The examples will be part of the PSS release
PORTABLE STIMULUS IN C++11
The UART Component in DSL

**Step 1:** Identify Behaviors
**Step 2:** Identify Data Flow
**Step 3:** Where do Actions Execute?
**Step 4:** What System Resources are required?

```
component uart_c {
    resource uart_r {...};
    pool [1] uart_r uart_p;
    bind uart_p {*};

    action read_in_a {
        output data_str data;
        lock uart_r u_busy;
    }

    action write_out_a {
        input data_str data;
        lock uart_r u_busy;
    }
}
```
UART Component in C++

**Step 1:** Identify Behaviors

**Step 2:** Identify Data Flow

**Step 3:** Where do Actions Execute?

**Step 4:** What System Resources are required?

```cpp
class uart_c : public component{
    PSS_CTOR(uart_c, component);
}

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

class read_in_a : public action {
    PSS_CTOR(read_in_a, action);
    output<data_str> data {"data"};
    lock<uart_r> u_busy{"u_busy"};
};
type_decl<read_in_a> ri_t;

class write_out_a : public action{
    PSS_CTOR(write_out_a, action);
    input<data_str> data {"data"};
    lock<uart_r> u_busy{"u_busy"};
};
type_decl<write_out_a> wo_t;
```
DMAC Component in DSL

- Step 1: Identify Behaviors
- Step 2: Identify Data Flow
- Step 3: Where do Actions Execute?
- Step 4: What System Resources are required?

```
class component dma_c {
  resource chan_r {...};
  pool [1] chan_r, chan_p;
  bind chan_p { * };

  action u2m_a {
    input data_str src_data;
    output mem_b dst_data;
    lock chan_r, chan;
  }

  action m2u_a {
    input mem_b src_data;
    output data_str dst_data;
    lock chan_r, chan;
  }

  action m2m_a {...}
}
```
The Steps to Creating a Portable Stimulus Model

• Step 1: Identify Behaviors
• Step 2: Identify Data Flow
• Step 3: Where do Actions Execute?
• Step 4: What System Resources are required?
Data Structures in DSL

- Portable Stimulus only cares about the structure of data
- Data type defines scheduling constraints
  - buffer = sequential
  - stream = parallel

```c
struct data_seg_s {
    rand int[4..1024] size;
    rand bit[63:0] addr;
};

buffer mem_b : data_seg_s {
};

enum data_dir_e {in, out};

stream data_str : data_seg_s {
    rand data_dir_e dir;
    rand int[1..2] stop_bits;
};
```
Portable Stimulus only cares about the structure of data

- Data type defines scheduling constraints
  - buffer = sequential
  - stream = parallel

```cpp
class data_seg_s : public structure {
    rand_attr<int> size {"size",
        range<int>(4,1024));
    rand_attr<bit> addr {"addr",
        width(63,0)};
};

class mem_b : public data_seg_s,
    public buffer {};

class data_dir_e : public enumeration {
    PSS_ENUM(data_dir_e,
        enumeration, in, out);
};

class data_str : public data_seg_s,
    public stream {
    rand_attr<data_dir_e> dir {"dir"};
    rand_attr<int> stop_bits {"stop_bits",
        range<int>(1,2)};
};
```
component pss_top {
  uart_c uart0;
  dma_c dma0;

  pool data_str stream_p;
  bind stream_p {*};
  pool mem_b mem_p;
  bind mem_p {*};

  action loopback_test {
    uart_c::read_in_a rd_i;
    uart_c::write_out_a wr_o;

    activity {
      rd_i;
      wr_o;
    }
  }
}
Top-Level Model in C++

class pss_top : public component {
    uart_c uart0 {"uart0"};
    dma_c dma0 {"dma0"};

    pool<data_str> stream_p {"stream_p"};
    bind b1 {stream_p};
    pool<mem_b> mem_p{"mem_p"};
    bind b2 {mem_p};

    class loopback_test : public action {
        action_handle<uart_c::read_in_a> rd_i {"rd_i"};
        action_handle<uart_c::write_out_a> wr_o {"wr_o"};

        activity a {
            rd_i,
            wr_o
        };
    };
};
GENERATING TESTS FROM PORTABLE STIMULUS
Consider Two Deployment Models

**UVM Testbench**

- PSS Model
- PSS Tool
- AXI
- UART
- DDR
- DMAC
- UVM Sequences

**SoC Testbench**

- PSS Model
- PSS Tool
- AXI
- UART
- DDR
- DMAC
- UVM Sequences
- C-test

© Accellera Systems Initiative

10/2/17
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() {
    ...
    // step N
    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

Could be SV or other language

10/2/17
Three Strategies for Implementing Exec Blocks

• Target Template: text templates for code generation

• Import Function: import and call external procedures/functions, assign to non random attributes

• Hardware/Software Interface: abstracted operations for registers, memory and interrupts
Target Template

```
action read_in_a {
  output data_str data;
  lock uart_r u_busy;
  constraint data.dir == in;

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

```
int main(int argc, char **argv) {

  // ...
  init_uart_rx(1); /* Moustache notation to refer to PSS fields */
  gen_uart_traffic(1, 128);
  // ...
  init_uart_rx(2);
  gen_uart_traffic(2, 27);
  return 0;
}
```
import void init_uart_rx(bit[1:0] stop_bits);
import void gen_uart_traffic(bit[1:0] stop_bits, int sz);
action read_in_a {
    output data_str data;
    lock uart_r u_busy;
    constraint data.dir == in;

    exec body {
        init_uart_rx( data.stop_bits );
        gen_uart_traffic( data.stop_bits , data.size );
    }
}

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;
}
Hardware/Software Interface Layer

import void gen_uart_traffic(bit[1:0] stop_bits, int sz);

action read_in_a {
    output data_str data;
    lock uart_r u_busy;
    constraint data.dir == in;
    Uart_regs uart_regs;

    exec body {
        uart_regs.lcr.stop_bits = data.stop_bits;
        gen_uart_traffic(data.stop_bits, data.size);
    }
}

int main(int argc, char **argv) {

    // ...
    write_reg(UART_LCR, 1 << 2);
    --
    gen_uart_traffic(1, 128);
    // ...
    write_reg(UART_LCR, 2 << 2);
    --
    gen_uart_traffic(2, 27);
    return 0;
}
Karthick Gururaj, Vayavya Labs

THE HARDWARE/SOFTWARE INTERFACE LIBRARY
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
What HSI Enables

- Stimulus Spec (Test intent)
- Test bench (in SV)
- Driver (in SV)
- DMAC (DUT)
  - RTL Simulation
  - Accelerated simulation with Emulator
  - FPGA, w/ external PCIe master
  - FPGA, w/ embedded core

- HSI Spec (Test realization)
- Test cases (in C, on host)
- Driver (in C, on host)
- DMAC (DUT)
- Test cases (in C, on host)
- Driver (in C, on target)
- Test cases (in C, on target)
- Driver (in C, on target)

HSI ensures Portability of test realization across Environments
What HSI Enables

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

- Registers
- FIFOs
- Virtual registers
- Descriptor management
- Interrupt management
- Sequences
- Device capabilities
Test Realization Using HSI

- All elements of HSI specification, including sequences, captured in C++
- HSI specification will integrate with test intent (in PSS-C++ or DSL)
- From HSI specification, driver APIs for desired environment/platform and language (C, SystemVerilog, etc) can be generated

- HSI specification currently under discussion in the WG, expect updates in upcoming revisions
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?

| Formalization of system level scenarios and models | Ability to formally describe coverage of the legal scenarios and attributes |

• 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
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
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 u2m_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;
        }
    }
}
```
CONCLUSION
We Hope You Learned...

- Portable stimulus is a high-value 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 (over 15 companies involved)
- 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...

• Provide feedback on the early adopter specification in review through 10/31/2017
  – The Early Adopter spec: http://www.accellera.org/downloads/drafts-review
  – Provide feedback on Accellera Portable Stimulus Forum:
    http://forums.accellera.org/forum/40-portable-stimulus-discussion/

• Participate in shaping this promising standard with your suggestions, use cases and requirements through:
  – your company’s Accellera representation
  – EDA vendor voicing your thoughts
  – reading the Early Adopter spec:
    http://www.accellera.org/downloads/drafts-review
  – providing feedback on Accellera Portable Stimulus Forum:
    http://forums.accellera.org/forum/40-portable-stimulus-discussion/
  – 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 India 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 this tutorial’s message and content