Agenda

• Whare are we here? – Tom Fitzpatrick, Siemens EDA
• Display Controller Example – Matan Vax, Cadence Design Systems
• Memory & Cache Examples – Adnan Hamid, Breker Verification Systems
• SoC Level Example – Hillel Miller, Synopsys
• Summary: IP to SoC & Post-Silicon – Tom Fitzpatrick, Siemens EDA
ASIC Biggest Functional Verification Challenge

- Creating sufficient tests to verify the design
- Knowing my verification coverage
- Managing the verification process
- Time to isolate and resolve a bug
- Time to discover the next bug
- Defining appropriate coverage metrics

Methodology Shifts Require New Thinking

- SystemVerilog brought a new approach to Verification
  - Standardized features from other proprietary languages
  - Directed testing → Constrained-Random

- C-R requires Functional Coverage to know what happened
- Needed narrow focus to build an Ecosystem
UVM Focused on How to Verify

- Primarily aimed at block level
- Modular reusable verification components
- Separate *What* from *How*
  - Composable Transaction-level sequences
  - Drivers/Monitors convert between transactions and signals
- Reuse abstract sequences with different agents
  - Sequences focus on *What*
  - Agents convert to protocol-specific *How*
- Each IP interface uses same approach
UVM Focused on How to Verify

- All tests/testbenches have same basic structure
- Horizontal Reuse
  - Same agent on different blocks with same interface
UVM Focused on How to Verify

- All tests/testbenches have same basic structure
- Horizontal Reuse
  - Same agent on different blocks with same interface
  - Block-level environments configured as components at next level
  - Block-level Sequences called from Virtual Sequences

- This was the right target 10 years ago
Verification is a Moving Target

• Block-level testing does not scale to SoC

• ASM:C == Gates:RTL == UVM:?
• Manually coded test usually in C
• No randomization for better coverage
• Manually coded IP libraries
  • No reuse from IPs
• Must procedurally model test space
  • Video data can come from various sources
  • There are N DMA channel
  • Graphics needs to power up before using it
  • Power management sequence
What If We Could Apply Automation to SoC Verification?

- Single Specification – Multiple tests
  - Formally describe test space to help tool generate tests conforming to system constraints
- Distributed test based on single specification
  - Reuse intent on processor
- Automated partitioning and coordination
- Constrained-random to find bugs at the scenario level
Stimulus at a Higher Level

- UVM Sequences define sets of transactions
  - Transaction contents randomized
  - Transaction flow not explicitly randomized
- Difficult to randomize between sequences

- Scenarios define sets of behaviors
  - Define critical verification intent
  - Define rules to support critical intent
- Simple PSS specification defines multiple scenarios
Portable Stimulus Brings Constrained-Random to Scenario Generation

- PSS partial specification defines critical verification intent
  - Don't have to "code and hope"
  - More intuitive and straightforward than SV functional coverage
- Rules allow tool to infer additional actions
- Allows random generation of scenarios
  - Each guaranteed to be legal
  - Constrain specific actions
  - Constrain scheduling relationships between actions
Key Aspects of Portable Stimulus

- Capture test intent
- Partial scenario description
- Composable scenarios
- Formal representation of test space
- Automated test generation
- Target multiple implementations

Separate test intent from implementation

High-coverage test generation across the verification process with much less effort
What is a Portable Stimulus Model?

The Abstract Model

• *What* does it do

The Realization Layer

• *How* does it do what it does
So What's the Point?

Activity defines critical behaviors
- May define *partial specification*
- Tools will *infer* additional elements to complete the scenario

Other parts of the model define how behaviors interact
- Instantiated components define available actions
- Flow object bindings constrain inference choices
- Resources constrain scheduling options
The Rubber Meets the Road

The Abstract Model must be implemented on different targets

Atomic Actions $\rightarrow$ target code
- Target code modeled in \textit{exec} blocks

\textit{Generator} assembles target code according to \textit{Activity} schedule
Generated Code Scheduled According to Activity
Generated Code Scheduled According to Activity
Agenda

• Where are we here? – Tom Fitzpatrick, Siemens EDA
• Display Controller Example – Matan Vax, Cadence Design Systems
• Memory & Cache Examples – Adnan Hamid, Breker Verification Systems
• SoC Level Example – Hillel Miller, Synopsys
• Summary: IP to SoC & Post-Silicon – Tom Fitzpatrick, Siemens EDA
Overview

• Purpose of this section:
  • Demonstrate the application of PSS to a typical IP verification problem
  • Explain the meaning and use of PSS resource pools and claims
  • Highlight a key advantage of modeling scenarios in PSS compared to SystemVerilog

• Structure of this section
  • A verification challenge – exercising data paths in a display controller
  • Modeling the scenario space in SystemVerilog and its limitations
  • Modeling the scenario space in PSS – the general solution
• 6 processing pipes:
  • Pipes 0..2 for video
  • Pipes 3..5 for graphics

• 4 overlay engines:
  • LCD0, LCD1, LCD2, and TV

• 5 interface engines
  • DSI_A, DSI_B, DP_A, DP_B, and HDMI
Display Controller Example Description (cont’)

- Each engine can process one data stream at a given time
  - 1..4 streams processed concurrently

- Datapath rules:
  - Video pipes cannot feed overlay engine LCD1
  - Graphics pipes cannot feed overlay engine LCD0
  - LCD0 can only feed DP_A
  - LCD1 can feed DSI_A or DSI_B
  - LCD2 can feed DSI_A, DSI_B or DP_B
  - TV can feed DP_A or HDMI
typedef enum {VID, GFX} pipe_kind_e;
typedef enum {LCD0, LCD1, LCD2, TV} overlay_kind_e;
typedef enum {DSI_A, DSI_B, DP_A, DP_B, HDMI} interface_kind_e;

class process_stream extends uvm_object;
  rand int pipe_id;
  rand pipe_kind_e pipe_kind;
  constraint pipe_kind_c {
    pipe_kind == VID -> pipe_id inside {0,1,2};
    pipe_kind == GFX -> pipe_id inside {3,4,5};
  }
  rand overlay_kind_e overlay_kind;
  rand interface_kind_e interface_kind;
  ...
endclass
Capturing Datapath Rules in SystemVerilog

Datapath rules:
- Video pipes cannot feed overlay engine LCD1
- Graphics pipes cannot feed overlay engine LCD0
- LCD0 can only feed DP_A
- LCD1 can feed DSI_A or DSI_B
- LCD2 can feed DSI_A, DSI_B or DP_B
- TV can feed DP_A or HDMI

```
class process_stream extends uvm_object;
...
constraint pipe2overlay_c {
  pipe_kind == GFX -> overlay_kind != LCD0;
  pipe_kind == VID -> overlay_kind != LCD1;
}
constraint overlay2interface_c {
  overlay_kind == LCD0 -> interface_kind == DP_A;
  overlay_kind == LCD1 -> interface_kind inside {DSI_A, DSI_B};
  overlay_kind == LCD2 -> interface_kind inside {DSI_A, DSI_B, DP_B};
  overlay_kind == TV -> interface_kind inside {DP_A, HDMI};
}
...
endclass
```
Modeling Parallel Streams in SystemVerilog

```systemverilog
class process_multi_stream extends uvm_object;
    rand process_stream streams[];
    constraint num_streams {
        streams.size inside {[1:4]};
    }

    constraint resource_unique_c {
        foreach (streams[i]) {
            foreach (streams[j]) {
                if (i != j) {
                    streams[i].pipe_id != streams[j].pipe_id;
                    streams[i].overlay_kind != streams[j].overlay_kind;
                    streams[i].interface_kind != streams[j].interface_kind;
                }
            }
        }
    }

endclass
```

Between one and four concurrent streams

 Pipes, overlays, and interfaces must be different for different streams

(SV/UVM code overheads – constructor, allocating arrays, uvm_object_utils...)

### streams[0]
- **pipe_id = 1**
- **overlay_kind = LCD2**
- **interface_kind = DSI_A**

### streams[1]
- **pipe_id = 2**
- **overlay_kind = LCD0**
- **interface_kind = DP_A**

### streams[2]
- **pipe_id = 2**
- **overlay_kind = LCD1**
- **interface_kind = DSI_B**
But How to Generalize?

• What have we achieved thus far?
  • Constraint model expresses relations between multiple concurrent streams
  • But this is just one specific scenario..

• What are we still missing?
  • Start / stop streams while processing of others is in progress
  • Balance workload across engines with tasks of different features/durations
  • Create random/complex schedules of streams
  • …

• The SystemVerilog model does not capture the true nature of the problem – tasks contending for limited resources
Modeling Resources in PSS

Resource object type, can include attributes and constraints

Pool contains N instances of a resource type

Built-in attribute ‘instance_id’ uniquely identifies the instance within a pool

User-defined enum attributed used to associate meaningful names with instances
Modeling a Single Data Stream

15.2 Claiming resource objects

Resource objects may be locked or shared by actions. This is expressed by declaring the resource reference field of an action. See Syntax 87, Syntax 88 and Syntax 89.

```plaintext
component display_c {
  ...
  action process_stream {
    lock pipe_r pipe;
    lock overlay_r overlay;
    lock interface_r interface;
    exec body {
      drive_stream(pipe.instance_id,
                   overlay.instance_id,
                   interface.instance_id);
    }
  }
}
```

Resources can be locked by actions, excluding other concurrent use.
Capturing Datapath Rules

Datapath rules:

- Video pipes cannot feed overlay engine LCD1
- Graphics pipes cannot feed overlay engine LCD0
- LCD0 can only feed DP_A
- LCD1 can feed DSI_A or DSI_B
- LCD2 can feed DSI_A, DSI_B or DP_B
- TV can feed DP_A or HDMI
Driving Parallel Streams

Parallel activities are guaranteed mutually exclusive resource assignments for locked resources

```
action process_multi_stream {
    rand int in [1..4] num_streams;
    activity {
        parallel {
            replicate (num_streams) {
                do process_stream;
            }
        }
    }
}
```
Creating Arbitrary Schedules

**action** random_schedule {
  rand int in [2..20] num_streams;
  activity {
    schedule {
      replicate (num_streams) {
        do process_stream;
      }
    }
  }
}

**action** my_scenario_26 {
  activity {
    parallel {
      do process_stream with { size == 60; };
      do random_schedule with {
        forall (p: process_stream) {
          p.size == 12;
        }
      };
    }
  }
}

Schedule all actions in a way that avoids resource conflicts

One long stream concurrent to multiple short streams
Summary

• PSS supports high-level modeling constructs, resource pools/claims among others
• These constructs naturally capture dependencies across behaviors in the design/test
• Thus, non-trivial flows and schedules can be easily described or randomized
• In contrast, SystemVerilog (and other verification languages) support only constrained randomization of data, hence modeling non-trivial flows/schedules is harder
Agenda

• Whare are we here? – Tom Fitzpatrick, Siemens EDA
• Display Controller Example – Matan Vax, Cadence Design Systems
• Memory & Cache Examples – Adnan Hamid, Breker Verification Systems
• SoC Level Example – Hillel Miller, Synopsys
• Summary: IP to SoC & Post-Silicon – Tom Fitzpatrick, Siemens EDA
Agenda

• Problem #1: DDR Memory Controller Page Management
• Problem #2: Traversing Cache Coherency state transitions
• Problem #3: Combine Problem #1 and Problem #2 in same scenario
• Scenario Modeling challenges with SV
• Modeling Executors, Memory, and Basic Read/Write tests
• Solution #1: Model DDR Page Management in PSS
• Solution #2: Model Cache Coherency state traversal in PSS
• Solution #3: Combining DDR Page Management and Cache Coherency scenarios
Many Possible Approaches to solving problem

• Each of Problem #1 or Problem #2 in isolation can be solved more simply

• This tutorial section shows two different approaches to creating sequences of operations and how they can be combined for cross coverage
The Problems

Problem #1: DDR Memory Controller page management

Problem #2: Cache Coherency state traversal
Problem #1: DDR Page management

• DDR is organized in Pages
  • Identified by group, bank, row, column

• Access in same page is fast
  • But memory controller may need to interleave refresh

• Access to different page in same bank is slow

• Objective: generate sequences of read/write addresses to stress memory controller page management
Problem #2: Cache Coherency State traversal

- SoC’s have multi-level caches
- Each cache level may have different coherency protocols

**Objective:** Generate sequences of operations scheduled on correct cores to traverse all coherency transitions
Problem #3: Combing Problem #1 & #2

- Traversing cache states must access DDR memory at various times
- DDR read/write operations have different timings based on pattern of address and page management.

**Objective:** Generate sequences of operations scheduled on correct cores to traverse all coherency transitions while stressing DDR Page management
Scenario Modeling Challenges with SystemVerilog

• Both Problem #1 DDR page management and Problem #2 Cache coherency state transitions requires sequences of related operations scheduled across different cpu cores.

• This is difficult to model natively in UVM/SystemVerilog

• PSS provides elegant solutions to such problems
Basic Read/Write tests

**Student Takeaway:** Quickly create random read/write tests

- Define how many executor cores are in the system
- Define available system memory
- Define basic read/write operations
- Define random address generator
- Put it all together
Configuring Executors & System Memory

```java
package config_pkg {
    const int NUM_CORES = 4;
}

struct core_traits_s : executor_trait_s {
    rand int in [0 .. config_pkg::NUM_CORES - 1] core_id;
}

component cores_c : executor_group_c<core_traits_s> {
    executor_c<core_traits_s> cores [config_pkg::NUM_CORES];
    exec init_down {
        foreach (c : cores[i]) {
            c.trait.core_id = i;
            add_executor(c);
        }
    }
}

component pss_top_rand_addrs_c {
    transparent_addr_space_c<> sys_mem;
    exec init_up {
        transparent_addr_region_s<> region;
        region.size = ( 1 << 40 );
        region.addr = 0x80000000;
        (void)sys_mem.add_region(region);
    }
}
```

Declare how many core are available in system
Each executor has a “trait” with its id
Array of cores with trait
Initialize Each core id
Declare available system memory
Basic Read/Write Operations: State Object

State Object will keep track of next memory address to operate on

Also stores a memory claim

Constrained to the next desired address

Constrained to native read/write sizes

```
state addr_s {
    rand bit[64] addr;
    rand transparent_addr_claim_s <> claim;
    constraint claim.addr == addr;
    constraint claim.size in [1,2,4,8];
}
```
Basic Read/Write Operations

```
component mem_ops_c {
    action write_a {
        input addr_s inp;
        rand executor_claim_s<core_traits_s> core;
        rand bit [64] data;
        exec body {
            addr_handle_t h = make_handle_from_claim(inp.claim);
            match (inp.claim.size) {
                [1]: write8(h, data[7:0]);
                [2]: write16(h, data[15:0]);
                [4]: write32(h, data[31:0]);
                [8]: write64(h, data[63:0]);
            }
        }
    }
    action read_a {
        input addr_s inp;
        rand executor_claim_s<core_traits_s> core;
        bit [64] data;
        exec body {
            addr_handle_t h = make_handle_from_claim(inp.claim);
            match (inp.claim.size) {
                [1]: data[7:0] = read8(h);
                [2]: data[15:0] = read16(h);
                [4]: data[31:0] = read32(h);
                [8]: data[63:0] = read64(h);
            }
        }
    }
} // mem_ops_c
```
Random address generator and instantiation

**Component: rand_addrs_c**

```plaintext
component rand_addrs_c {
    action rand_addrs_a {
        output addr_s out;
    }
}
```

**Output an unconstrained random address**

**Component: rand_addr_test_c**

```plaintext
component rand_addr_test_c {
    rand_addrs_c rand_addrs;
    mem_ops_c mem_ops;
    pool addr_s addr_p;
    bind addr_p *;

    action rand_addr_test_a {
        activity {
            do rand_addrs_c::rand_addrs_a;
            select {
                do mem_ops_c::write_a;
                do mem_ops_c::read_a;
            }
        }
    }
}
```

**Instantiate and bind**

**Generate next address**

**Do read or write**
Basic Memory Tests

```
component pss_top_rand_addrs_c {
  transparent_addr_space_c<> sys_mem;
  exec init_up {…}
  cores_c cores;
  rand_addr_test_c rand_addr_test[10];
  action entry_a {
    activity {
      schedule {
        replicate (100){
          do rand_addr_test_c::rand_addr_test_a;
        }
      }
    }
  }
}
```

Declare available system memory & cores

Instantiate

Schedule actions
Solution #1: DDR Page Management

Solution #1: DDR Memory Controller page management
Solution #1: DDR page management

- DDR is organized in Pages
  - Identified by group, bank, row, column
- Access in same page is fast
  - But memory controller may need to interleave refresh
- Access to different page in same bank is slow

**Objective**: generate sequences of addresses to stress memory controller page management
Modeling DDR Memory Bank Address Sequences

**Student Takeaway:** Modeling address sequences with state objects

```plaintext
component ddr_page_addrs_c {
  state ddr_page_s {
    rand bit [ 2] group;
    rand bit [ 2] bank;
    rand bit [16] row;
    rand bit [10] column;
    rand bit [64] addr;

    constraint addr[18: 9] == column;
    constraint addr[34:19] == row;
    constraint addr[36:35] == bank;
    constraint addr[38:37] == group;
  }

  pool ddr_page_s ddr_page_p;
  bind ddr_page_p *;
}
```

- State Objects allow sequencing constraints via `prev` variable
- DDR page identification
- Resolved address
- Memory type specific constraints
- Create pool and bind
Modeling DDR Memory Bank Address Sequences

**Hit on same page:**
fast, but may need refreshes

**Hit on same bank, different page:**
page switching is slow

**Dictionary of other address picking strategies**

```
action ddr_same_page_a {
    output ddr_page_s ddr_page;

    constraint c {
        ddr_page.group == ddr_page.prev.group ;
        ddr_page.bank == ddr_page.prev.bank ;
        ddr_page.row == ddr_page.prev.row ;
        ddr_page.column == ddr_page.prev.column ;
    }
}

action ddr_same_bank_a {
    output ddr_page_s ddr_page;

    constraint c {
        ddr_page.group == ddr_page.prev.group ;
        ddr_page.bank == ddr_page.prev.bank ;
        ( ddr_page.row != ddr_page.prev.row ||
          ddr_page.column != ddr_page.prev.column );
    }
}

action ddr_next_col_a {
    output ddr_page_s ddr_page;

    constraint c {
        ddr_page.group == ddr_page.prev.group ;
        ddr_page.bank == ddr_page.prev.bank ;
        ddr_page.row == ddr_page.prev.row ;
        ddr_page.column == (ddr_page.prev.column + 1) & 0x3ff ;
    }
}
action select_strategy_a {
  activity {
    select {
      [8]: do ddr_same_page_a;
      [1]: do ddr_next_col_a;
      [1]: do ddr_same_bank_a;
      ...
    }
  }
}

action constrain_addr_a {
  input ddr_page_s ddr_page;
  output addr_s out;
  constraint out.addr == ddr_page.addr;
}

// component ddr_page_addrs_c

component ddr_test_c {
  ddr_page_addrs_c ddr_page_addrs;
  mem_ops_c mem_ops;
  pool addr_s addr_p;
  bind addr_p *;

  action ddr_test_a {
    activity {
      do ddr_page_addrs::select_strategy_a;
      do ddr_page_addrs::constrain_addr_a;
      select {
        do mem_ops::write_a;
        do mem_ops::read_a;
      }
    }
  }
}
Modeling DDR Memory Bank Address Sequences

component pss_top_ddr_addrs_c {
  cores_c cores;
  ddr_test_c ddr_test[10];
  transparent_addr_space_c<> sys_mem;
  exec init_up {...}
}

action entry_a {
  activity {
    schedule {
      replicate (100){
        do ddr_test_c::ddr_test_a;
      }
    }
  }
}

Instantiate

Declare available system memory

Schedule actions

![Diagram showing memory bank address sequences with actions like read_a, write_a, ddr_page_s, constrain_addr_a, ddr_next_col_a, ddr_same_page_a.](image)
Solution #2: Cache Coherency State traversal

Problem #2: Cache Coherency state traversal
Solution #2: Cache Coherency State Transitions

• SoC’s have multi-level caches
• Each cache level may have different coherency protocols

• **Objective:** Generate sequences of operations scheduled on correct cores to traverse all coherency transitions
Modeling Cache Coherency State Transitions

**Student Takeaway:** Modeling coherency sequences with action inferencing

**component** coherency_c {
    **enum** cl_state_e {INVALID, EXCLUSIVE, MODIFIED, OWNED, SHARED};

    **state** cl_state_s {
        **rand** cl_state_e cl_state;
        constraint initial -> cl_state == INVALID;
        **rand** int home_core_id;
        **rand** int count;
        constraint initial -> count == 0;
    }

    **pool** cl_state_s cl_state_p;
    **bind** cl_state_p *;
}

**action** reset_counter_a {
    output cl_state_s out;
    constraint out.count == 0;
}

- Track target cache state
- "Home" core_id for scenario
- Test length counter
- Instantiate & bind
- Action to reset count
Modeling Cache Coherency State Transitions

abstract action transition_base_a {
  input cl_state_s inp;
  output cl_state_s out;

  constraint out.count == inp.count +1;
  constraint out.home_core_id == inp.home_core_id ;
}

action invalid_to_exclusive_a : transition_base_a {
  constraint transition {
    inp.cl_state == INVALID;
    out.cl_state == EXCLUSIVE;
  }
  activity {
    do mem_ops_c::read_a with {
      core.trait.core_id == this.inp.home_core_id; }
  }
}

action exclusive_to_invalid_a : transition_base_a {
  constraint transition {
    inp.cl_state == EXCLUSIVE;
    out.cl_state == INVALID;
  }
  activity{
    do mem_ops_c::write_a with {
      core.trait.core_id != this.inp.home_core_id; }
  }
}

// coherency_c

Input & output the state
Track scenario length
Track “home” core id
Prev/Next State
Read from “home” core
Write from “other” core
Modeling Cache Coherency State Transitions

component pss_top_coherency_simple_c {
    cores_c cores;
    rand_addrs_c rand_addrs;
    mem_ops_c mem_ops;
    coherency_c coherency;
    pool addr_s addr_p;
    bind addr_p *;

    transparent_addr_space_c<> sys_mem;
    exec init_up {...}

    action entry_a {
        activity {
            do rand_addrs_c::rand_addrs_a;
            do coherency_c::exclusive_to_invalid_a
                with { out.count == 20; };
        }
    }
}
Modeling Cache Coherency State Transitions

```plaintext
component coherency_c {
  ...
  action invalid_to_modified_a : transition_base_a { ... }
  action invalid_to_owned_a : transition_base_a { ... }
  action invalid_to_shared_a : transition_base_a { ... }
  action exclusive_to_modified_a : transition_base_a { ... }
  action exclusive_to_shared_a : transition_base_a { ... }
  ...
  action state_selector_a {
    rand int count;
    activity {
      select {
        do exclusive_to_invalid_a with {out.count == this.count;};
        do modified_to_invalid_a with {out.count == this.count;};
        do owned_to_invalid_a with {out.count == this.count;};
        do shared_to_invalid_a with {out.count == this.count;};
      }
    }
  }
}
```

Define all transitions

`count` variable will be constrained from above

Always end in invalid state so that following scenarios can reuse address
Modeling Cache Coherency State Transitions

```c
component coherency_test_c {
    rand_addrs_c rand_addrs;
    mem_ops_c mem_ops;
    coherency_c coherency;
    pool addr_s addr_p;
    bind addr_p *;

    action coherency_test_a {
        rand int count;

        activity {
            do rand_addrs_c::rand_addrs_a;
            do coherency_c::reset_counter_a;

            do coherency_c::state_selector_a
                with {count == this.count};
        }
    }
}
```

- Instantiate & bind
- `count` variable will be constrained from above
- Pick rand address & reset scenario length counter
- Infer random sequence of coherency transitions
modeling cache coherency state transitions

component pss_top_coherency_c {
    cores_c cores;
    coherency_test_c coherency_test[10];
}

transparent_addr_space_c<> sys_mem;
exec init_up {...}

action entry_a {
    activity {
        schedule {
            replicate (100) {
                do coherency_test_c::coherency_test_a
                    with {count == 20;};
            }
        }
    }
}

force a chain of 20 coherency transitions

declare available system memory

instantiate

each inference chain of random transitions reuses one random address
Solution #3: Combing Solution #1 & #2

- Traversing cache states must access DDR memory at various times
- DDR read/write operations have different timings based on pattern of address and page management.

**Objective:** Generate sequences of operations scheduled on correct cores to traverse all coherency transitions while stressing DDR Page management
Combining Problem #1 and Problem #2

**Student Takeaway:** Composition of scenarios

```cpp
component coherency_ddr_test_c {
    ddr_page_addrs_c ddr_page_addrs;
    mem_ops_c mem_ops;
    coherency_c coherency;
    pool addr_s addr_p;
    bind addr_p *;

    action coherency_ddr_test_a {
        rand int count;

        activity {
            do ddr_page_addrs_c::select_strategy_a;
            do ddr_page_addrs_c::constrain_addr_a;

            do coherency_c::reset_counter_a;

            do coherency_c::state_selector_a
                with {count == this.count};
        }
    }
}
```

- **Instantiate & bind**
- `count` variable will be constrained from above
- Pick DDR bank address
- Reset scenario length counter
- Infer random sequence of coherency transitions
Combining Problem #1 and Problem #2

```cpp
component pss_top_coherency_ddr_c {
  cores_c cores;
  coherency_ddr_test_c coherency_ddr_test[10];
  transparent_addr_space_c<> sys_mem;
  exec init_up {...}
}

action entry_a {
  activity {
    schedule {
      replicate (100) {
        do coherency_ddr_test_c::coherency_ddr_test_a
        with {count == this.count};
      }
    }
  }
}
```

- Instantiate
- Declare available system memory
- Force a chain of 20 coherency transitions
- Each inference chain of random transitions reuses item from DDR address sequence
Summary

• Problem #1: Modeled DDR page address sequences using state variables

• Problem #2: Modeled Coherency State transitions using action inferencing

• Problem #3: Combined problem #1 & #2 using model composition

• Difficult and time consuming to model this in SV or C/C++

• Elegant solutions in PSS
Agenda

• Where are we here? – Tom Fitzpatrick, Siemens EDA
• Display Controller Example – Matan Vax, Cadence Design Systems
• Memory & Cache Examples – Adnan Hamid, Breker Verification Systems
• SoC Level Example – Hillel Miller, Synopsys
• Summary: IP to SoC & Post-Silicon – Tom Fitzpatrick, Siemens EDA
Agenda

• Chaining SOC Scenarios – Recap from DVCON 2020 – 5 Minutes
• Modeling DMA Pipelining Scenario – 10 Minutes
• Generalizing Modeling IP Pipelining Scenarios – 5 Minutes
• Modeling Chaining/Pipelining SOC Scenarios – 5 Minutes
Multiple IP Scenario Intent

CHAINING SCENARIO

CPU WRITE → MEMORY → DMA TRANSFER → MEMORY → DISPLAY SHOW

DMA TRANSFER → MEMORY → CPU READ

SIMPLE SOC

CPU → DMA → MEMORY → DISPLAY
A Block-to-System Portability and Productivity – Recap from DVCON 2020

Verification productivity goal #2: Composing multiple Stimulus Chaining from different IP Stimulus
System Level Definition

```c
extend component pss_top {
   // RTL Agents
   dma_c dma;
   lte_c lte;
   display_c display;
   ...

   // Execution agents
   pool [4] execution_agent_r cpu;
   pool [1] lte_vip_r lte_vip;
   ...
}
```

```c
extend component pss_top {
   // Address Space
   contiguous_addr_space_c<mem_trait_s> mem_addr_space;
   addr_region_s<mem_trait_s> dram_region;
   addr_region_s<mem_trait_s> flash_region;
   exec init {
      dram_region.trait.kind = DRAM;
      mem_addr_space.add_region(dram_region);
      mem_addr_space.add_region(flash_region);
   }
}
```
Chaining Stimulus from Multiple IP through Memory Buffers

Common Flow Object Type to Declare Output Buffer

Chaining Structures: Sequential, Parallel, Graphs

Usage of Storage Allocation for Data Integrity

Chaining Coverage
package common_target {
    buffer data_buffer {
        rand addr_space_pkg::addr_claim_s<mem_trait_s> mem_seg;
    }

    abstract action mem_copy_a {
        input data_buffer buf_in;
        output data_buffer buf_out;
    }
}
IP Owner’s Stimulus

Source:
External VIP ingress to Memory
Memory2Memory:
(a) Memory to External VIP egress.
(b) External loop back VIP ingress to Memory.
Sink:
Memory to External VIP

Source:
External VIP to Memory

Sink:
Memory to External VIP

Source:
External VIP ingress to Memory
Memory2Memory:
(a) Read Memory
(b) Write Memory
IP Owner’s Action

```plaintext
action <device-name>_source_a {
    output data_buffer ctb_out;
    ...
}
action <device-name>_mem2mem_a :
    mem_copy_a {...}

action <device-name>_sink_a {
    input data_buffer ctb_in;
    ...
}

action <device-name>_source_a {
    output data_buffer ctb_out;
    ...
}

action core_mem2mem_a :
    mem_copy_a {...}
```
Sequential Chaining

```plaintext
action sequential_chaining_a {
    activity {
        // Source
        select {
            [10] : do lte_source_a;
            [20] : do cdma_source_a;
            [10] : do camera_source_a;
        }
        // Memory2Memory
        replicate (2) {
            select {
                do core_mem2mem_a;
                do dma_mem2mem_a;
                do bluetooth_mem2mem_a;
            }
        }
        // Sink
        do display_sink_a;
    }
}
```
Parallel Chaining

extend component pss_top {
  action entry {
    schedule {
      replicate (5) {
        do sequential_chaining_a;
      }
    }
  }
}
Chaining Combinations Coverage

covergroup chaining_cg {
    source: coverpoint source_id;
    mem2mem_0: coverpoint mem2mem_id0;
    mem2mem_1: coverpoint mem2mem_id1;
    sink: coverpoint sink_id;
    size: coverpoint size;
    cross chain: source,
                  mem2mem_0,
                  mem2mem_1,
                  sink_id,
                  size;
}

action sequential_chaining_a {
    rand source_e source_id;
    rand mem2mem_e mem2mem_id0, mem2mem_id1;
    rand sink_e sink_id;
    rand int size;
    activity {
        // Source
        select {
            [10] : do lte_source_a with {
                size == this.size;
                this.source_id == LTE_SOURCE;
            }
            [20] : do cdma_source_a with {
                size == this.size;
                this.source_id == CDMA_SOURCE;
            }
            [10] : do camera_source_a with {
                size == this.size;
                this.source_id == CAMERA_SOURCE;
            }
        }
        // Memory2Memory
        ...
        // Sink
        ...
    }
}
Controlling and Covering the interleaving of streaming scenario’s

**Simple stream example with interleaving**

DMA Pipeline (2 steps)
1. Move (Denoted Mx, where x is, the transfer number the step belongs to)
2. Wait (Denoted Wx), consumes time depending on the attributes (e.g. size).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0</td>
<td>M0</td>
<td>M0</td>
<td>M0</td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>W0</td>
<td>M1</td>
<td>M1</td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>M1</td>
<td>M2</td>
<td>W1</td>
<td></td>
</tr>
<tr>
<td>W0</td>
<td>W1</td>
<td>W2</td>
<td>M2</td>
<td></td>
</tr>
<tr>
<td>W1</td>
<td>M2</td>
<td>W1</td>
<td>W0</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
<td>W0</td>
<td>W2</td>
<td></td>
</tr>
</tbody>
</table>

Controlling and Covering such scenarios in a modular and portable framework is needed for many Verification/Validation tasks.

Using single controller (e.g. embedded core, AXI bus), need to manage the interleaving N DMA stream transfers. For example, On the left we show 4 possible scenarios for 3 DMA transfers.

- **Scenario 0**, uses the most DMA channels, could find corner case bugs
- **Scenario 3**, would work if only 2 DMA channels were available
- **Scenario 2**, may provide best performance results, for specific DMA transaction types
Naive implementation of DMA interleaved stream scenario’s in SV

while (j<NOF_SCENARIOS) begin
  // Randomly pick a new step at a time, accounting for transfer steps that
  // have already been picked. There is no obvious set of constraints that
  // can guarantee: (1) Exactly 2 steps will be picked for each transfer
  // (2) first step will appear before second step for all transfers.
  while (i<NOF_TRANSFERS*2) begin
    if (available(move) and available(wait))
      step = $random % 2; // 2 available steps
    else if (available(move))
      step = 0;
    else
      step = 1;
    scenario_step[i].type=step;
    scenario_step[i].transfer_id=get_available_transfer_id(step);
    i++;
  end
  if (!exists(scenario_step) begin
    add_scenario(scenario_step);
    j++;
  end
end

Algorithm is not efficient in providing different scenario’s

Controlling the scenarios, requires changes in functions used in pseudo code

Would need to have a different implementation if number of steps grew.

Cannot mix with other stimulus using some of the same resources

Cannot port code to embedded C implementation, need to rewrite

Controlling, Adjusting, Mixing, Covering, Porting, scenarios in PSS is done in a declarative way
Modeling DMA stream interleaving in PSS

```pascal
package dma_pkg {
  const int NOF_TRANSFERS = 4;
  state state_s {
    rand bit move [NOF_TRANSFERS] ;
    rand bit wait [NOF_TRANSFERS] ;
    constraint initial -> {
      foreach (move[i]) {
        move[i] == 0;
        wait[i] == 0;
      }
    }
  }
}

action move_a {
  input state_s in_s;
  output state_s out_s;
  rand int transfer_num;
  constraint out_s.move[transfer_num] == 1'b1;
  constraint {
    foreach(out_s.move[i]) {
      out_s.wait[i] == in_s.wait[i];
      if (i != transfer_num) {
        out_s.move[i] == in_s.move[i];
      }
    }
  }
}

action wait_a {
  input state_s in_s;
  output state_s out_s;
  rand int transfer_num;
  constraint out_s.wait[transfer_num] == 1'b1;
  constraint {
    foreach(out_s.wait[i]) {
      out_s.move[i] == in_s.move[i];
      if (i != transfer_num) {
        out_s.wait[i] == in_s.wait[i];
      }
    }
  }
}
```

- **Initializes state object fields**
- **Marks that MOVE was done**
- **Constrains other state values not to change**
- **Marks that WAIT was done**
- **Constrains other state values not change**
Scheduling DMA steps with interleaving enabled

```verilog
component dma_c {
    import ip0_pkg::*;
    pool state_s state_p;
    bind state_p *;
    action dma_transfer_a {
        rand int transfer_num;
        move_a M;
        wait_a W;
        activity {
            M with {transfer_num == this.transfer_num; };
            W with {transfer_num == this.transfer_num; };}
    }
    action all_dma_transfers_a {
        dma_transfer_a DT[NOF_TRANSFERS];
        activity {
            schedule {
                replicate (i:NOF_TRANSFERS) {
                    DT[i] with {transfer_num == i;};
                }
            }
        }
    }
}
```

Schedule operator will interleave M, W across multiple transfers
Easy to over constrain

<table>
<thead>
<tr>
<th>action move_a</th>
</tr>
</thead>
<tbody>
<tr>
<td>input state_s in_s;</td>
</tr>
<tr>
<td>output state_s out_s;</td>
</tr>
<tr>
<td>rand int transfer_num;</td>
</tr>
<tr>
<td>constraint out_s.move[transfer_num] == 1'b1;</td>
</tr>
<tr>
<td>constraint {</td>
</tr>
<tr>
<td>foreach(out_s.move[i]) {</td>
</tr>
<tr>
<td>out_s.wait[i] == in_s.wait[i];</td>
</tr>
<tr>
<td>if (i != transfer_num) {</td>
</tr>
<tr>
<td>out_s.move[i] == in_s.move[i];</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>action wait_a</th>
</tr>
</thead>
<tbody>
<tr>
<td>input state_s in_s;</td>
</tr>
<tr>
<td>output state_s out_s;</td>
</tr>
<tr>
<td>rand int transfer_num;</td>
</tr>
<tr>
<td>constraint out_s.wait[transfer_num] == 1'b1;</td>
</tr>
<tr>
<td>constraint {</td>
</tr>
<tr>
<td>foreach(out_s.wait[i]) {</td>
</tr>
<tr>
<td>out_s.move[i] == in_s.move[i];</td>
</tr>
<tr>
<td>if (i != transfer_num) {</td>
</tr>
<tr>
<td>out_s.wait[i] == in_s.wait[i];</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

extend action wait_a |
| constraint countones(in_s.move) == NOF_TRANSFERS; |

Waits can only start if all the moves are done
Pipelining resources across streams that interleave

How to model stimulus for IPs that pipeline tasks into multiple sub-tasks and require interaction with stimulus.

Modeling multistep stimulus models for IP, that enables interleaving the different sub-tasks.

Modeling resources that are shared across different streams when relinquished at different stages of the pipeline.

Coverage of interesting interleaving scenarios, when there are multiple in-flight tasks.

**Examples**
- DMA IP that has multiple channels resources, with sub-tasks to move and wait for completion. Same channel needs to be used for a DMA transfer for both MOVE and WAIT.
- DISPLAY IP that has PIPE, OVERLAY and INTERFACE resources, with sub-tasks to handle each resource.
Pipelining Stimulus – PSS Modeling

Usage of schedule operator together with state flow objects in model to enable generation and characterization of pipelining scenarios.

Name actions as: stream_step_<i>_a
Where <i> is 0 to the Number of sub-tasks, minus 1, required for the stream (e.g. DMA has 2 sub-tasks, DISPLAY has 3).

Usage of resource object needed for each sub-task.

Modeling and Sampling Covergroups to characterize the interleaving of a generated test.

Add additional constraints over input state objects to control pipelining.
Modeling 3 step DISPLAY pipeline

```plaintext
package display_pkg {
  resource display_engine_step0_r {}
  resource display_engine_step1_r {}
  resource display_engine_step2_r {}

  state state_s {
    rand bit step0 [NOF_TRANSFERS] ;
    rand bit step1 [NOF_TRANSFERS] ;
    rand bit step2 [NOF_TRANSFERS] ;
    rand bit [NOF_TRANSFERS] step0_b;
    rand bit [NOF_TRANSFERS] step1_b;
    rand bit [NOF_TRANSFERS] step2_b;
    constraint {foreach([step0[i]]) {
      step0_b[i] == step0[i];
      step1_b[i] == step1[i];
      step2_b[i] == step2[i];
    }}
    constraint initial -> {
      foreach (step0[i]) {
        step0[i] == 0;
        step1[i] == 0;
        step2[i] == 0;
      }
    }
  }
}

component display_c {
  action stream_step0_a {
    input state_s in_s;
    output state_s out_s;
    rand int transfer_num;
    lock display_engine_step0_r engine_l;
    constraint out_s.step0[transfer_num] == 1'b1;
    constraint {
      foreach(out_s.step0[i]) {
        out_s.step1[i] == in_s.step1[i];
        out_s.step2[i] == in_s.step2[i];
        if (i != transfer_num) {
          out_s.step0[i] == in_s.step0[i];
        }
      }
    }
    covergroup {
      step0: coverpoint in_s.step0_b;
      step1: coverpoint in_s.step1_b;
      step2: coverpoint in_s.step2_b;
      all: cross step0, step1, step2 {
        ignore_bins interleaving = all with ( (step0 | step1) == step0) &&
          (step1 | step2) == step1)
      }
    }
  }
}
```

Lock resource based on step, for display first resource is PIPE, second is OVERLAY and third is INTERFACE.

Sample cross of the states at every step, goal is to achieve all interleaving's characterized by this cross. See slide below for more details.
component display_c {
  import display_pkg::*;
  pool state_s state_p;
  bind sample_state_p *;

  action stream_a {
    rand int transfer_num;
    stream_step0_a s0;
    stream_step1_a s1;
    stream_step2_a s2;
    activity {
      s0 with {transfer_num == this.transfer_num;}
      s1 with {transfer_num == this.transfer_num;}
      s2 with {transfer_num == this.transfer_num;}
    }
  }

  action all_stream_a {
    stream_a s[NOF_TRANSFERS];
    activity {
      schedule {
        replicate (i:NOF_TRANSFERS) {
          s[i] with {transfer_num == i;}
        }
      }
    }
  }
}
Modeling multiple pipelined interleaved streams, where resources are shared when they relinquished from their step

```plaintext
extend component soc_c {
  import display_pkg::*;
  pool [6] display_engine_step0_r display_engine_step0_p;
  bind display_engine_step0_p *;
  pool [4] display_engine_step1_r display_engine_step1_p;
  bind display_engine_step1_p *;
  pool [5] display_engine_step2_r display_engine_step2_p;
  bind display_engine_step2_p *;
  action all_ip_all_tasks_a {
    activity {
      schedule {
        replicate (i:DISPLAY_STREAMS) {
          do display_c::all_stream_a with {comp == pss_top.soc.display[i];};
        }
      }
    }
  }
}
```

Pools of DISPLAY resources can be shared over multiple controllers, when a controller relinquishes a resource, another can acquire, no need to wait for the completion of the stream that had the resource.
Applying resource constraints from display example to generic stream interleaving pipeline pattern

```plaintext
package display_pkg {
  enum pipe_kind_e {VID, GFX};
  extend resource display_engine_step0_r {
    rand pipe_kind_e kind;
    constraint {
      kind == VID -> instance_id in [0..2];
      kind == GFX -> instance_id in [3..5];
    }
  }
  enum overlay_kind_e {LCD0, LCD1, LCD2, TV};
  extend resource display_engine_step1_r {
    rand overlay_kind_e kind;
    constraint instance_id == int(kind);
  }
  enum interface_kind_e {DSI_A, DSI_B, DP_A, DP_B, HDMI};
  extend resource display_engine_step2_r {
    rand interface_kind_e kind;
    constraint instance_id == int(kind);
  }
}

extend action display_c::stream_a {
  constraint pipe2overlay_c {
    s0.engine_l.kind == GFX -> s1.engine_l.kind != LCD0;
    s0.engine_l.kind == VID -> s1.engine_l.kind != LCD1;
  }
  constraint overlay2interface_c {
    s1.engine_l.kind == LCD0 ->  s2.engine_l.kind == DP_A;
    s1.engine_l.kind == LCD1 ->  s2.engine_l.kind in [DSI_A, DSI_B];
    s1.engine_l.kind == LCD2 ->  s2.engine_l.kind in [DSI_A, DSI_B, DP_B];
    s1.engine_l.kind == TV   ->  s2.engine_l.kind in [DP_A, HDMI];
  }
}
```
Coverage of interleaving scenario

```vhdl
covergroup {  
  step0: coverpoint in_s.step0_b;  
  step1: coverpoint in_s.step1_b;  
  all: cross step0, step1  
  {  
    ignore_bins interleaving = all with ((step0 | step1) != step0)  
  }  
} cg;
```
SOC Scenarios combining pipeline tasks from multiple IPs

```cpp
component riscv_c {
  ip0_c ip0;
  ip1_c ip1;
  action all_ip_all_trx_a {
    activity {
      parallel {
        do ip0_c::all_simple_a;
        do ip1_c::all_simple_a;
      }
    }
  }
}
```

Interleaving scenarios of different IP run independent of each other
action simple_a {
    input buf in_buf;
    output buf out_buf;
    rand int transfer_num;
    simple_step_0_a s0;
    simple_step_1_a s1;
    simple_step_2_a s2;
    activity {
        s0 with {transfer_num == this.transfer_num; }
        s1 with {transfer_num == this.transfer_num; ;}
        s2 with {transfer_num == this.transfer_num; ;}
    }
}

Chaining all Pipelining
Summary of Chaining and Interleaving Scenarios

• Showed how to take IP models and simply apply to SOC, using the chaining example.
• Showed how to get fine grained control over interleaving /pipelining stream scenario using schedule together with state flow object.
• For interleaving/pipelined stream example, showed how PSS declarative constructs are used and how it would be very difficult or impossible to do it declaratively in SV.
• Showed how easy it is to combine different types of SOC scenarios, for example combining chaining and interleaving/pipelining stream scenarios.
Agenda

• Whare are we here? – Tom Fitzpatrick, Siemens EDA
• Display Controller Example – Matan Vax, Cadence Design Systems
• Memory & Cache Examples – Adnan Hamid, Breker Verification Systems
• SoC Level Example – Hillel Miller, Synopsys

• Summary: IP to SoC & Post-Silicon – Tom Fitzpatrick, Siemens EDA
PSS Summary

• Formal IP test space specification
  • Can be created early in IP lifecycle as a documentation of test space

• Formal system scenario documentation
  • Can be created at SoC architecture definition time

• Quick composition of complex multi-IP test scenarios
  • Automated handling of state, resource and scheduling dependencies in test

• Partial test scenario
  • Create test without deep understanding of complete SoC

• Test generation time and runtime coverage reports
  • Coverage for power state transitions, functional modes etc.

• Tests are portable
  • From IP (UVM/SystemC) to embedded processor on SoC to post-silicon
Formal test space specification

- Display controller base PSS model defines its test space completely
- Workhorse action can be used to build IP or SoC test scenarios
- Rudimentary knowledge of display needed to create SoC tests

```
extend component pss_top {
    display_c display_0;
    display_c display_1;

    action soc_scenario {
        activity {
            schedule {
                do display_c::process_stream;
                do display_c::process_stream;
            }
        }
    }
}
```

```
component display_c {
    pool [6] pipe_r pipe_pool;
    pool [4] overlay_r overlay_pool;
    pool [5] interface_r interface_pool;
    pool display_state_s display_state;

    // Initialization
    action init_display {
        output display_state_s display_state;
    }

    action display_base {
        input display_state_s display_state;
        constraint display_state.initial != true;
    }

    // Workhorse action
    action process_stream : display_base {
        lock pipe_r pipe;
        lock overlay_r overlay;
        lock interface_r interface;
    }
}
```
Conclusion

PSS is a declarative, formal, portable test specification leading to automated generation of platform-specific tests.

PSS 2.0 is here and production ready.

Great ideas for PSS 2.1 and beyond.

Come join us to shape the PSS future.
Thank you!

Any questions?