2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Using Portable Stimulus to Verify an ARMv8 Sub-System SoC Integration

Mike Baird, Willamette HDL

Aileen Honess, Breker Verification Systems

accellera SYSTEMS INITIATIVE

2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Agenda

- Brief Introduction to PSS – Aileen & Mike
- PSS modeling concepts – Mike
- Overview of key PSS constructs – Mike
- ARMv8 integration verification - Aileen

accellera SYSTEMS INITIATIVE

Why Waste Your Life Writing and Debugging Tests?

UVM

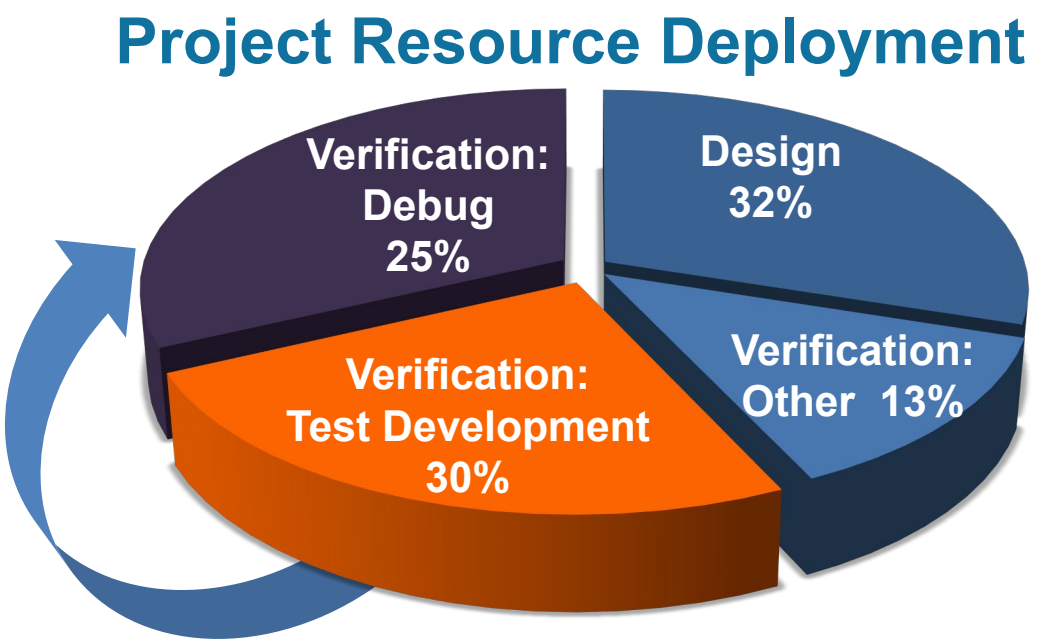
Laborious concurrent sequence, scoreboard and coverage authoring, limited reuse

SoC

Time-consuming, manual C tests targeting multi-core platforms with many corner-cases

Silicon

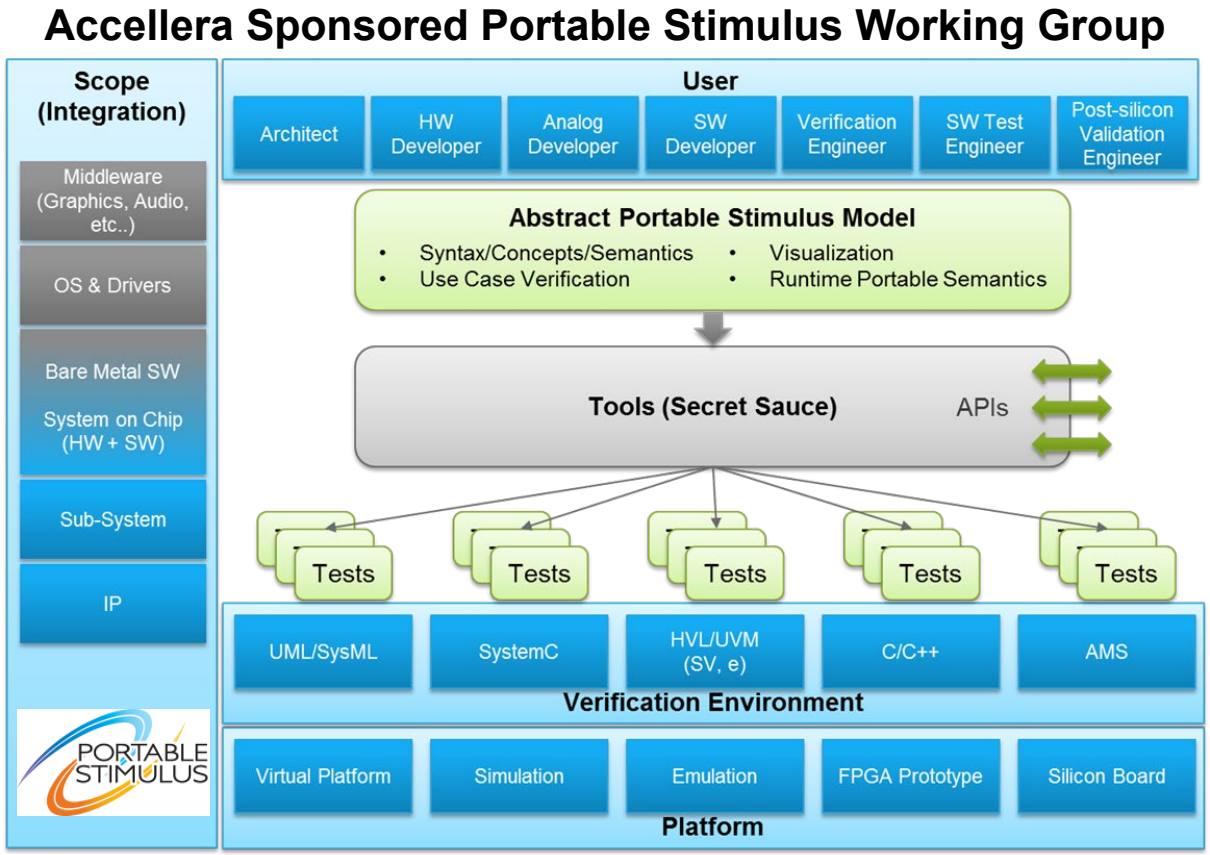
Complex diagnostic patterns with no link to verification, limited visibility



Test development drives debug

Complex directed test cases are hard to get right

Accellera Portable Stimulus Standard



Proposed Portable Stimulus Specification (Courtesy: Accellera Systems Initiative)

- Abstract, specification-driven testing
- Designed to be portable across:
  - Verification process phases
  - Verification platforms
  - Engineering groups
- The real win:
  - Eliminate UVM painful coding
  - Create intricate SDV corner cases
  - Automate silicon diagnostics
- PSS 1.0 powerful…
  but the tools make the difference!

PSS Concept: Flight Booking Example

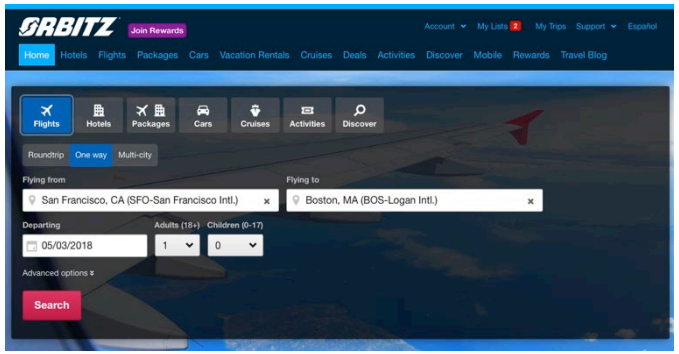
Traditional

Contact lots of airlines for lots of flights

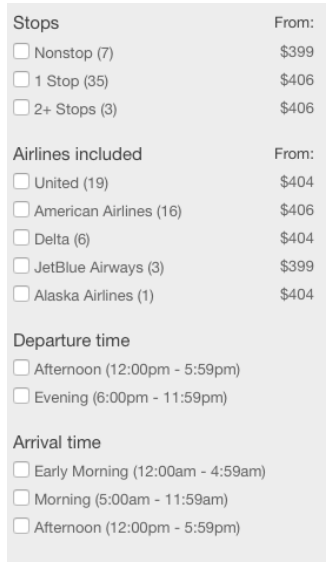
Is a bit like

Authoring lots of tests

Modern

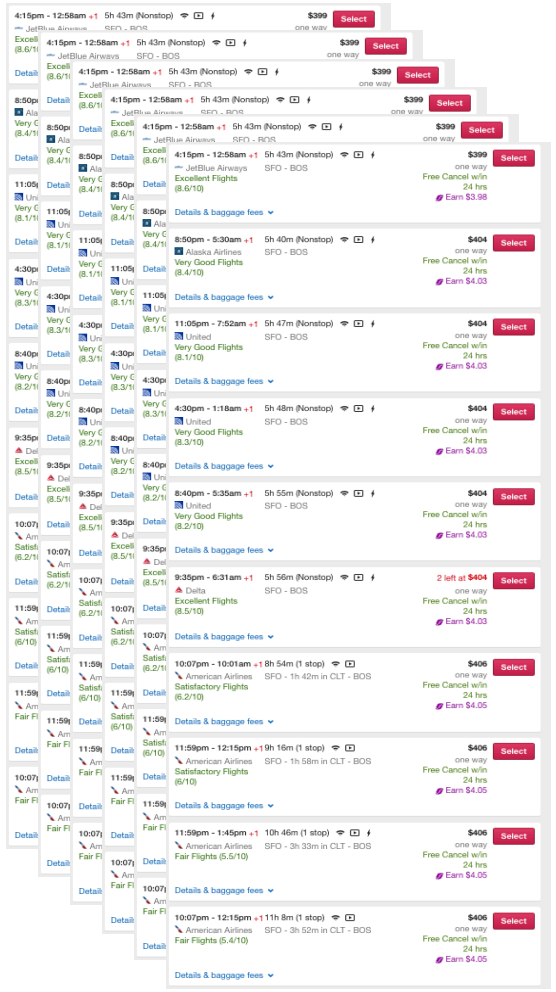


Describe intent



Set constraints

Options synthesized

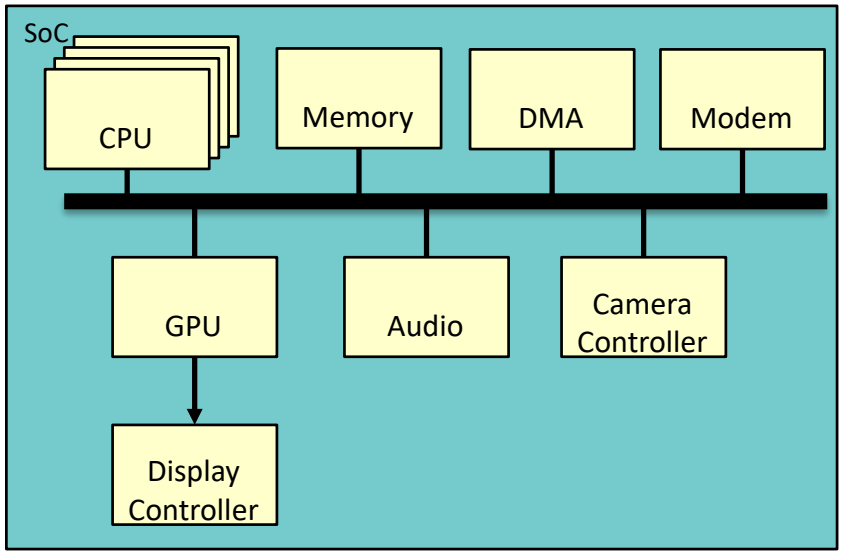


PSS Language Flavors

- Two "flavors" or different syntax for PSS
  - Domain Specific Language (DSL) syntax
    - SystemVerilog like syntax
  - C++ using a C++ class library that is semantically equivalent to the DSL
- Today
  - PSS language explained using DSL
  - ARMv8 verification using C++

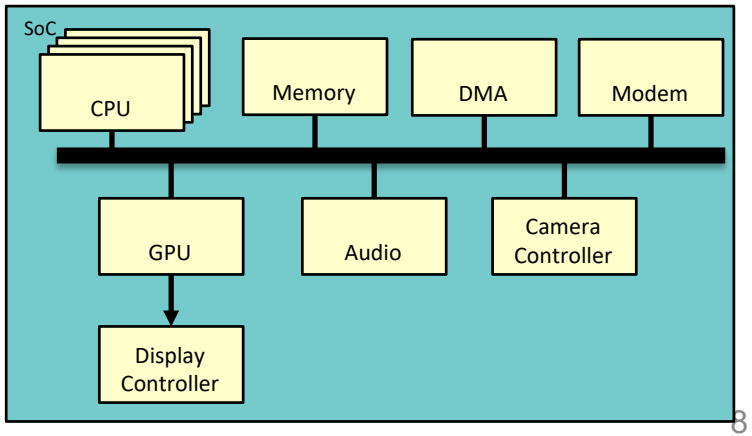
Test Scenarios and the Scenario Space

- *Test scenario (or scenario)*
  - High-level documentation of a use case
    - "Tells" a story
      - "Capture an image, manipulate it with a photo processor and save it to memory"
      - "Capture audio and transmit it out the modem"
  - Performed to ensure end-to-end functionality
    - Looks at the system as a whole – not just individual parts
- Test scenarios are derived from "user stories"
- The *scenario space* encompasses the possible test scenarios or use cases for a particular system



PSS Scenario Model

- PSS language is used to model the the scenario space of a system
  - AKA a PSS scenario model
- Tools can "process" the PSS language scenario model and represent it in a graph-based scenario model
  - Tool solves for one or multiple test scenarios from this model
    - Test case(s) are generated for a target test environment
- It can be useful to think of a PSS scenario model as an "abstract" layer or model or on top of an underlying test layer or model
  - Underlying layer:
    - UVM tests/model
    - "C" based tests/model

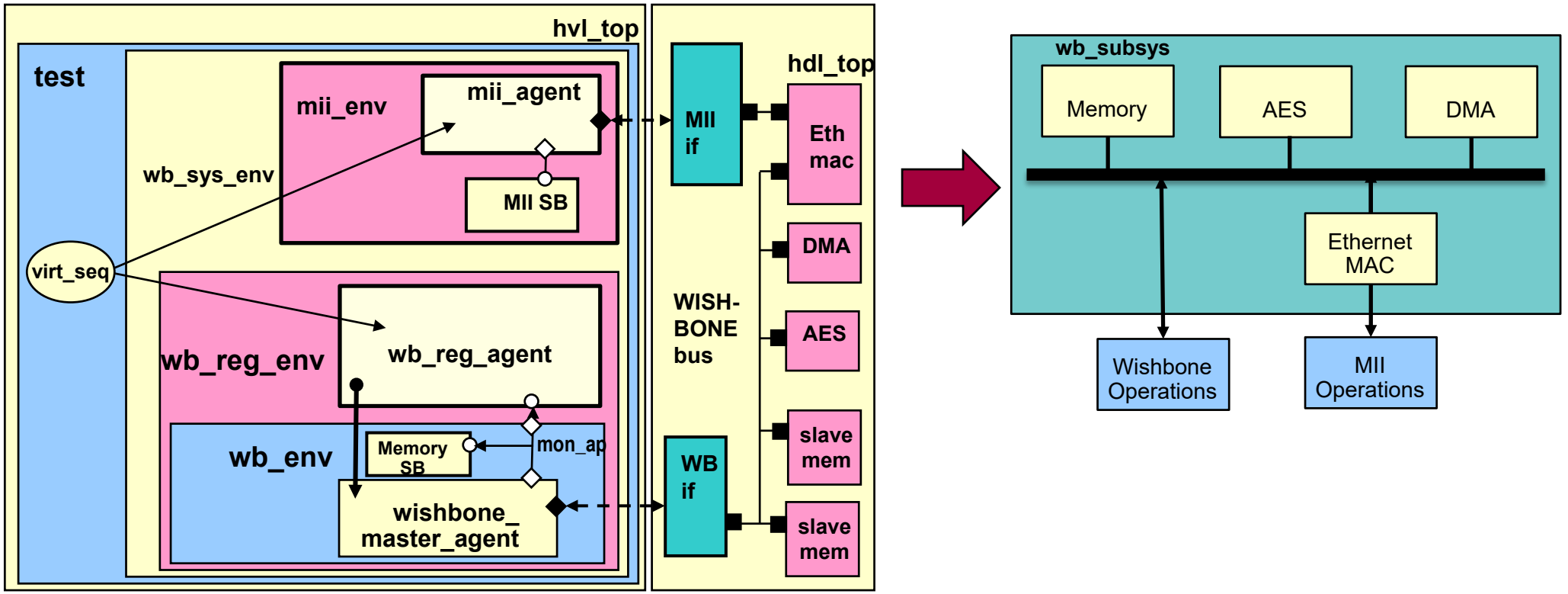


What then is a PSS Scenario Model?

- PSS models the scenario space in terms of
  - Resources
    - What is available to accomplish scenarios
    - CPU, DMA, Encrypt/decrypt, Graphics processor, camera etc.
  - Actions
    - Behaviors of a scenario
    - Encrypt/decrypt, transmit/receive, image capture, dma transfer etc.
  - Data and control flows
    - Information flow in the scenario
    - Buffers, streams, states etc.

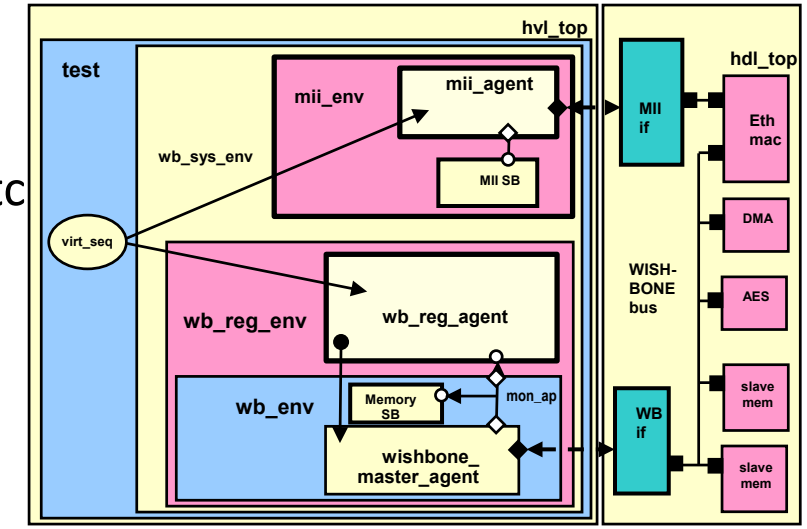
Paradigm Shift

- Coming from a UVM or Software Driven Verification (SDV) environment to PSS requires a paradigm shift (a must have "aha!" moment)
  - You will struggle mightily until you make this shift
- Must move from a "testbench viewpoint" to a "test intent viewpoint"



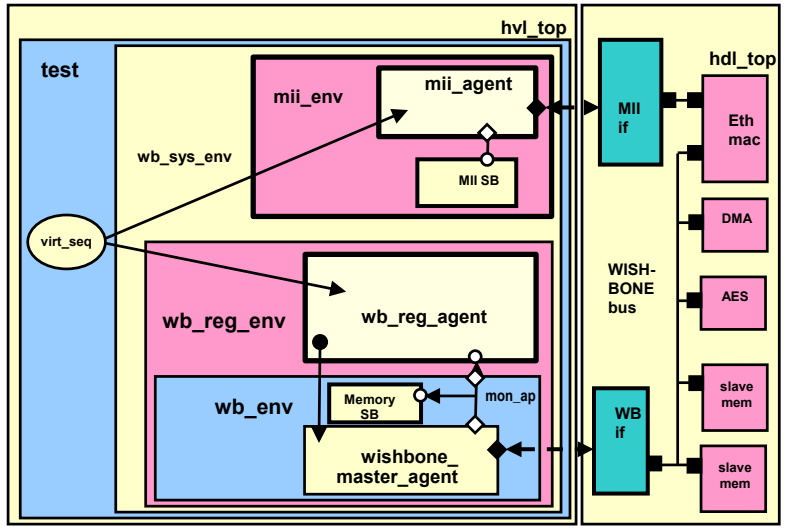
"Testbench Viewpoint"

- Think in terms of what the testbench must do to cause a desired behavior in the system (DUT)
  - Look at the pieces of the system as boxes with some kinds of interfaces that are exercised to cause DUT behaviors
  - Write code that executes on VIP/Processor that uses the DUT interface to exercise and observe DUT behaviors
    - UVM sequences, C functions
      - Initialize IPs
      - Cause DMAs, encrypt/decrypt, ethernet transfers etc
      - Do all the interfaces operations
      - Get results
      - "Scoreboard" DUT behaviors

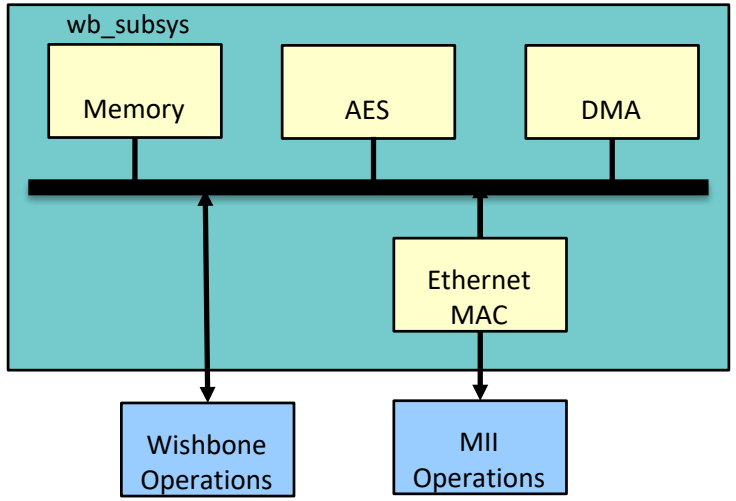


Testbench Example

- Focusing just on the DMA IP in the `wb_subsys` example...
- Write sequences/functions
  - DMA initialization
    - Write/read memory mapped registers to initialize the DMA
  - DMA transfer
    - Initialize buffer(s)
    - Start transfer
    - Verify transfer
- Write a test (and another and another...)
  - Code is written with a specific execution platform in mind
    - Sequences for a UVM VIP interface agent
    - C code for a processor
  - Explicit calls to sequences/functions that execute on the VIP agent/processor
  - Written from the point of view *"What do I need to do to the DMA IP?"*



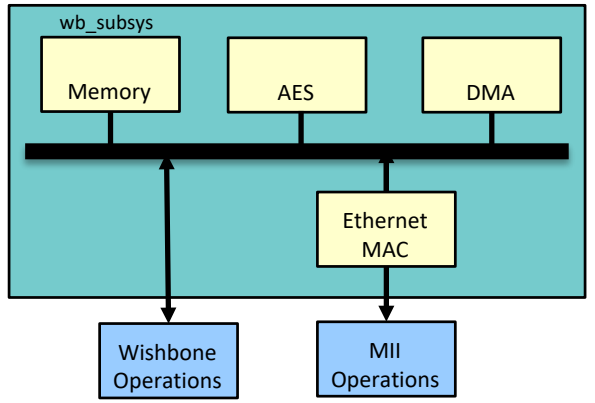
Test Intent Viewpoint



- Think in terms of what the system does
  - *What does it do?*
  - *What are its behaviors?*
  - *What inputs does it require?*
- Write a PSS model that captures the test *intent*
  - Not the test *implementation* – we are *not* writing tests in PSS
- Model (scenario model) of the test intent
  - Describes what the system must do to prove it has been verified
    - Is as abstract as possible to make tests re-targetable
  - Describes the system in terms of resources, requirements and behaviors
  - Partial description
    - E.g. what the requirements are, not how they are met
      - Let the PSS tool infer the "how they are met"

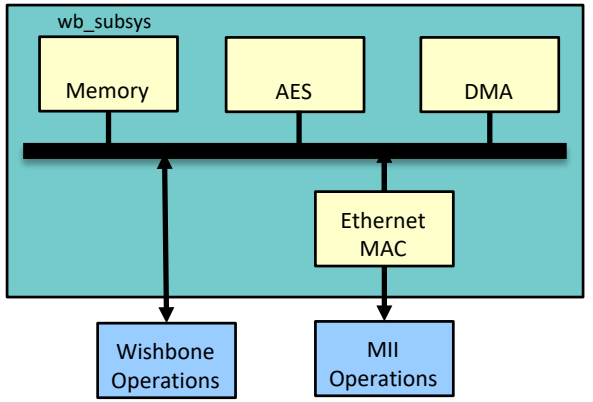
Test Intent Example - Capture

- Focusing just on the DMA IP in the `wb_subsys` example...
- Capture the behaviors, resources and requirements
  - DMA initialization
    - Perform the DMA configuration
  - DMA transfer
    - Perform DMA
    - Require a DMA channel as a resource
    - Require source and destination memory blocks
  - PSS written from the point of view: *"What does this DMA IP do?*



Test Intent Example - Inference

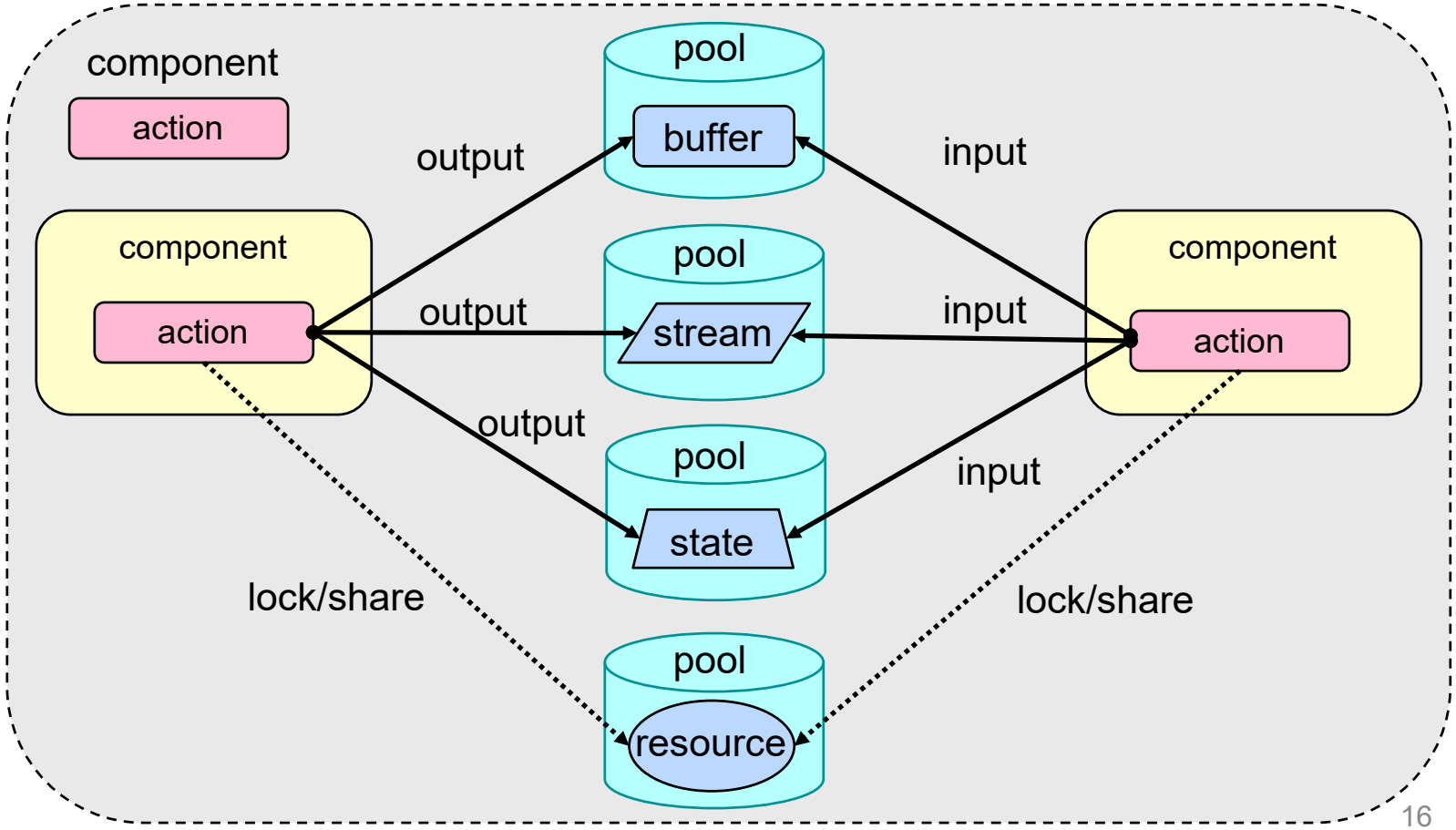
- Tool infers a source for required resources
  - DMA initialization
    - VIP/Processor that executes the DMA configuration
  - DMA transfer
    - What DMA channel is used
    - What provides the source block
      - Other IP in the system - Ethernet MAC, AES etc.
      - VIP/processor
    - What infers the DMA transfer to happen
      - Other IP in the system - Ethernet MAC, AES etc.
      - VIP/processor



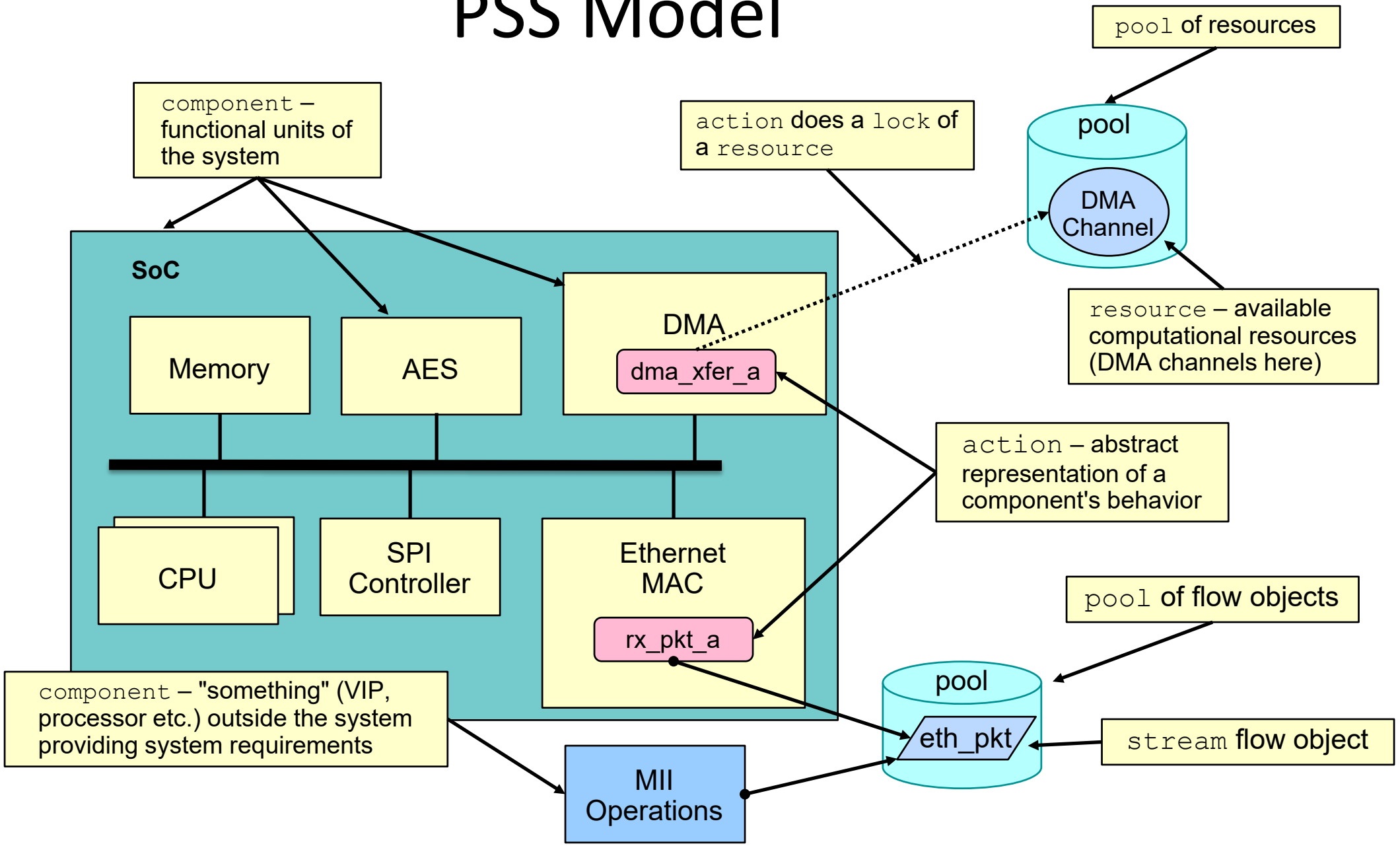
Capture PSS Model → Compose scenario → Tool inference and Visualization → Test generation for target testbench environment

PSS Overview - Constructs

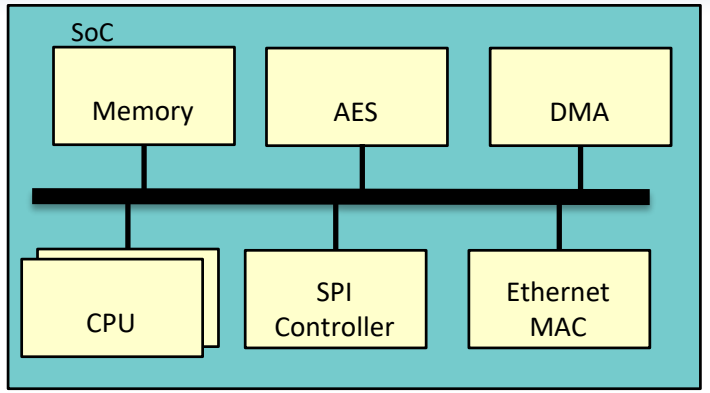
- component
- Flow objects: buffer, stream, state
- action
- resource
- Pools of flow objects and resources



PSS Model


pool of resources
component – functional units of the system
action does a lock of a resource
pool
DMA Channel
resource – available computational resources (DMA channels here)
SoC
Memory
AES
DMA
dma_xfer_a
action – abstract representation of a component's behavior
CPU
SPI Controller
Ethernet MAC
rx_pkt_a
pool of flow objects
pool
eth_pkt
stream flow object
component – "something" (VIP, processor etc.) outside the system providing system requirements
MII Operations

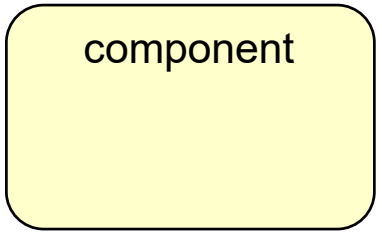
Components



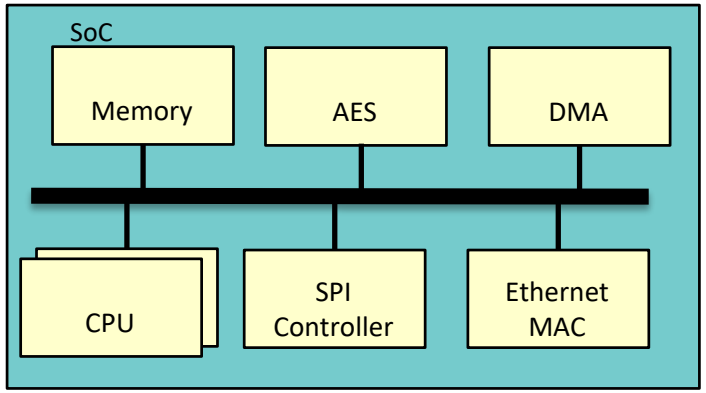
- Abstract representation of the functional units of a system
  - HW IPs
  - HW Cores
  - Testbench VIP
  - The DMA, AES, GPX etc. in the SoC diagram would be components in a PSS model
- Components are containers
  - Instances of other components
  - Actions
  - Resources

```
component component_name{}
```

```
Example:
component dma_c{ ... };
```



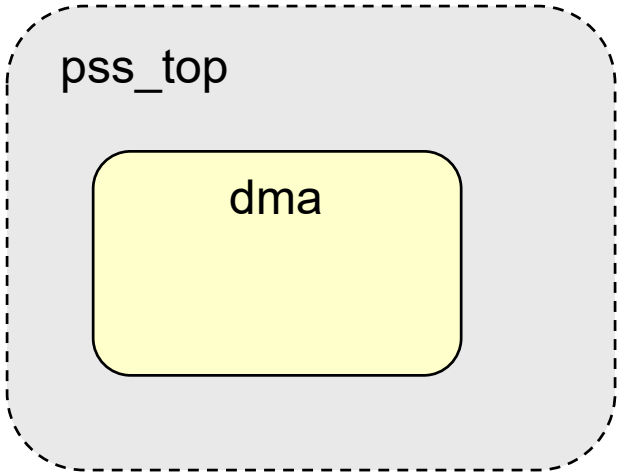
Component Instances



- Components may contain instances of other components
  - Creates a hierarchical structure
  - The top or root component
    - `pss_top`

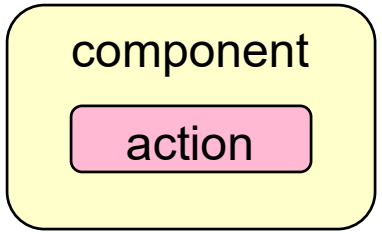
```
component_type instance_name;
```

```
Example:
pss_top {
...
  dma_c dma{};
...
}
```



Modeling Behaviors - Actions

- Actions
  - Defined in a component
  - Abstract representation of component behavior
    - Transmit a packet, DMA transfer, capture video etc.
- Compound actions
  - "Call" other actions
    - May be scheduled in any order but are sequentially by default
      - Various operators covered later for more complex scheduling of actions
- Atomic actions
  - "Call" test code that is one of
    - C code that would run on a target processor
    - SV code that runs on SV or UVM VIP
    - Other target languages



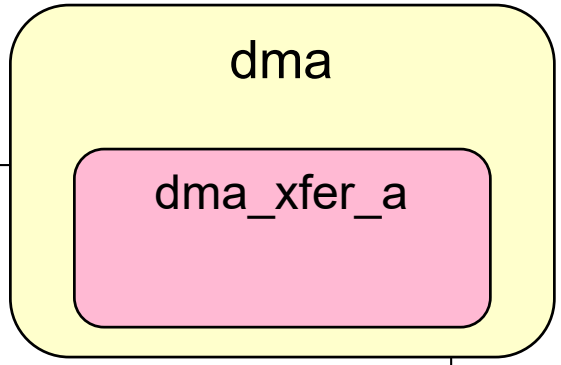
Defining Atomic Actions

- Atomic actions contain test code in a block referred to as an exec block
  - `body` exec block contains either of
    - Literal C or SV code (we will use this type for now)
    - Imported or exported function calls (we will illustrate later)
  - There are other exec block types available but not covered here

```
action action_name {
  exec body C """
    // target language code
  """;
}
```

Example:

```
component dma_c {
  action dma_xfr_a {
    exec body C """
     printf("\n *** dma_xfr_a action ***\n\n");
    """;
  }
  ...
}
```



Action Inputs and Outputs

- An action is an *abstract representation* of component behavior
- Actions may require inputs
  - A DMA transfer requires data to move
  - An encrypt requires data and a key to encrypt
- Actions may generate outputs
  - A DMA transfer generates data that was moved
  - An encrypt generates encrypted data
- The input of an action could be the output of another action
  - This is a key abstraction of PSS (matching inputs and outputs)
  - The properties of the inputs must be agreed upon by all involved actions
    - Size, format or direction of data
    - Location in memory of data

Flow Objects

- Flow objects are the abstract representation of the input and output information of an action
- PSS has 3 flow object types
  - Buffer
    - Represents *persistent* data
  - Stream
    - Represents *transient* data
  - State
    - Represents state information

Buffers

buffer

- Represent *persistent* data (data storage) that can be written and read
  - Data once generated is always available
  - Typically represent data or control buffers in memories
- Schedule dependency
  - A `buffer` must be written (generated) before it is read

```
buffer name { body_item, ... }
```

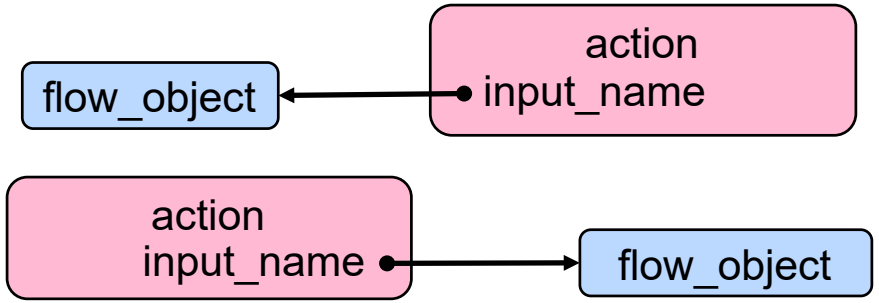
```
Example:
buffer mem_buff {
  rand bit[31:0] addr;
  rand bit[15:0] size;
}
```

mem_buff

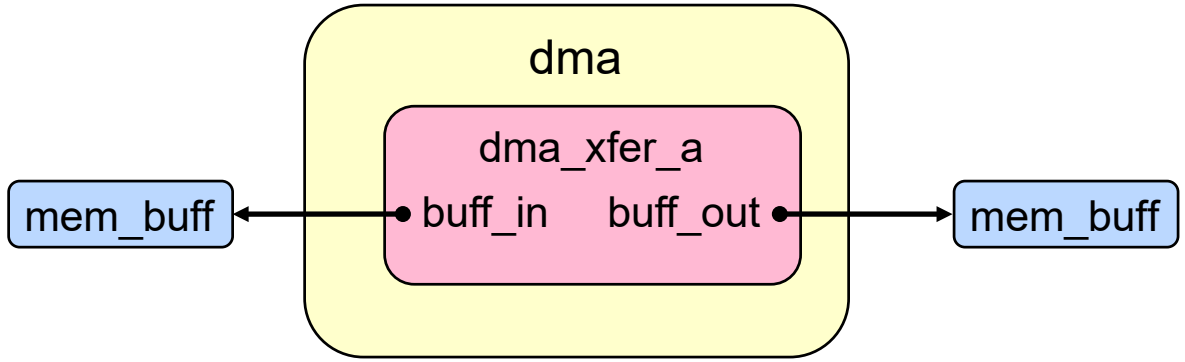
Action Inputs and Outputs

- Actions may define the inputs they require
- Actions may generate outputs

```
input  flow_object_type input_name;

output flow_object_type output_name;
```




```
Example:
component dma_c {
  action dma_xfer_a {
    input  mem_buff buff_in;
    output mem_buff buff_out;
    ...
  }
  ...
}
```



Streams

stream

- Streams represent *transient* information
  - Typically represents data flow, message or control exchange
    - Typically models the transmission of data or control
- Schedule dependency
  - Streams are exchanged between actions that are *concurrent*
- Examples
  - "Transmit" of an ethernet packet from an Ethernet MAC to Ethernet VIP
  - "Receive" of a packet by a modem from VIP

```
stream name { body_item, ... }
```

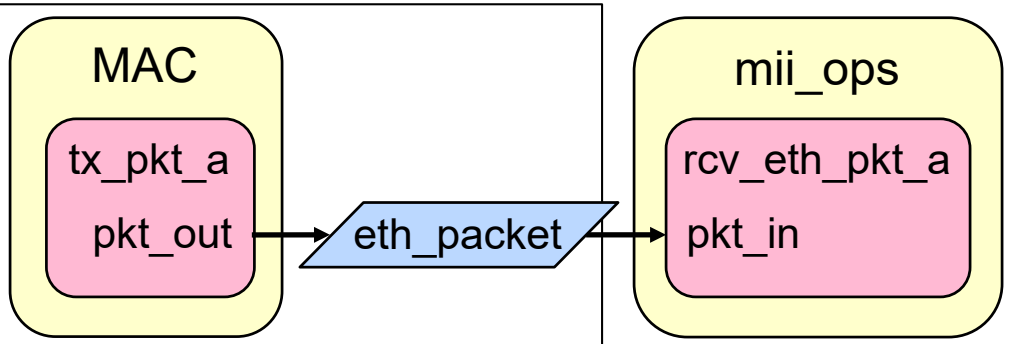
eth_packet

```
Example:
stream  eth_packet {
  rand bit [15:0] payload_len;
  rand bit [47:0] dest_MAC_addr;
  rand bit [47:0] srce_MAC_addr;
}
```

Stream Action inputs/outputs Example

- Transmit packet action of MAC has a `stream` output
- Receive packet action of MII Operations has a `stream` input

```
Example:
stream  eth_packet {
    rand bit [15:0] payload_len;
    rand bit [47:0] dest_MAC_addr;
    rand bit [47:0] srce_MAC_addr;
  }

component mac_c {
  action tx_pkt_a {
    output eth_packet pkt_out; // output ethernet packet
    ...
  }
  ...
}

component mii_ops_c {
  action rcv_tx_eth_pkt_a {
    input eth_packet pkt_in; // input ethernet packet
    ...
} ... }
```



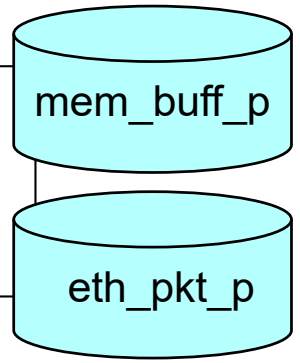
actions must execute in parallel

Flow Object Pools

- Flow object pools are collections of flow objects (`buffer, stream, state`)
- Actions use a pool to exchange flow objects
  - An action's inputs and outputs are references to a flow object pool

```
pool flow_object_type_name  pool_name;
```

```
Example:
pool mem_buff   mem_buff_p; // pool of mem_buff

pool eth_packet eth_pkt_p;  // pool of ethernet packets
```



Flow Object Pool Binding

- *Every* flow object resides in some pool
- Every action of an instance of a component
  - Outputs objects to or inputs objects from a specific pool
- Pool `bind` directives determine which pool is accessible to each action in each component instance
- Two forms of binding
  - Default binding – associate a pool by object type
    - `bind` pool to any action's input or output of the object type
  - Explicit binding – associate a pool with a specific action's input or output of the object type (not discussed here)

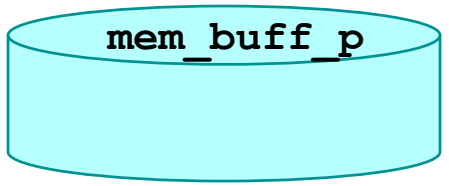
```
// bind pool to any action's input or output of pool_name type
bind pool_name {*};
```

Flow Object Pool Binding Example

- Pool binding: `mem_buff` pool to any action with a `mem_buff` input or output

Example:

```
component pss_top {
  pool mem_buff mem_buff_p; // pool of mem_buff
  // bind pool to any action's input/output of type mem_buff
  bind mem_buff_p {*};
  ...
}

component dma_c {
  action dma_xfer_a {
    input  mem_buff buff_in;
    output mem_buff buff_out;
    ...
  }
  ...
}
```



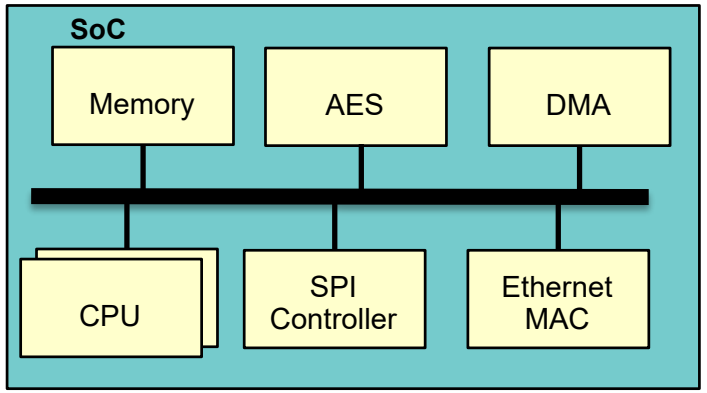
action inputs and outputs are bound to `mem_buff_p`

Resource Objects (Resources)

- Resource objects represent available computational resources that may be associated with actions
  - I.e. resources describe what is available in the execution environment to accomplish a scenario
- Resources relate to the underlying model IPs, buses etc.
  - In the diagram below we might list DMA channels, CPU, GPX, Ethernet MAC, USB device, Encrypt/decrypt engine and the camera as resources

```
resource resource_name { body_item, ... }
```

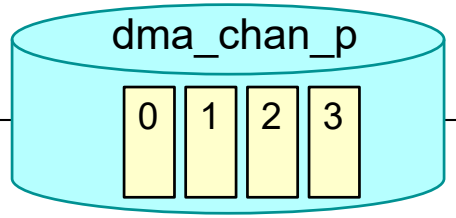
```
Example:
component dma_c {
  resource dma_chan_r {} // DMA channel resource
  ...
}
```



Resource Pools

- Resource object pools are collections of objects of a `resource` type
- Pool size (total number of resources)
  - Default size is 1, may be set to any size
- Resources may be claimed by actions
  - `lock`
    - An action claims an available resource
    - This action has exclusive use of the resource throughout its execution

```
pool        type_name name;
pool[size]  type_name name;
```

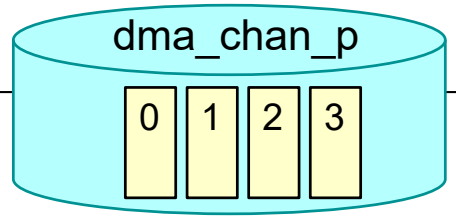



```
Example:
component dma_c {
  resource dma_chan_r {} // DMA channel resource
  pool[4]  dma_chan_r dma_chan_p; // pool of DMA channels, size 4
  ...
}
```

Resource Pool Binding

- Every resource object resides in some pool
- Every action of an instance of a component can be assigned a resource of a certain pool
- Like flow object pools, `bind` directives determine which pool is accessible to each action in each component instance
- Same types of binding (default and explicit)

```
bind pool_name {*};
bind pool_name *; // equivalent syntax
```



Example:

```
component dma_c {
  resource dma_chan_r {} // DMA channel resource
  pool[4]  dma_chan_r dma_chan_p; // pool of DMA channels, size 4
  bind dma_chan_p {*};   // bind pool to anything that uses a dma_chan_r
  ...
}
```

Constraints

- Actions and flow objects may have constraints applied
  - Defines legal combinations of data and control resources
  - **Key abstraction in PSS**, limits the possible scenario solution space
    - A valid PSS scenario is one that satisfies ALL constraints

```
constraint constraint_expression;
constraint constraint_name { constraint_expression; ... }
```

Example:

```
component dma_c {
...
  dma_xfr_a {
    input  mem_buff   buff_in;  // source of DMA
    output mem_buff   buff_out; // dest of DMA
    lock   dma_chan_r dma_chan; // lock a DMA channel
    constraint buff_in.size < 4096;  // constrain size of DMA xfer
    // constrain output buffer to same size as input buffer
    constraint buff_out.size == buff_in.size;
    ...
  } }
```

Provider (source) of `mem_buff` *must* meet these constraints

Packages

- PSS package is similar to a package in SV or a namespace in C++
- Package
  - Defines a namespace (or scope)
  - A namespace for declarations
    - Data flow types, resource types, enumerations etc.

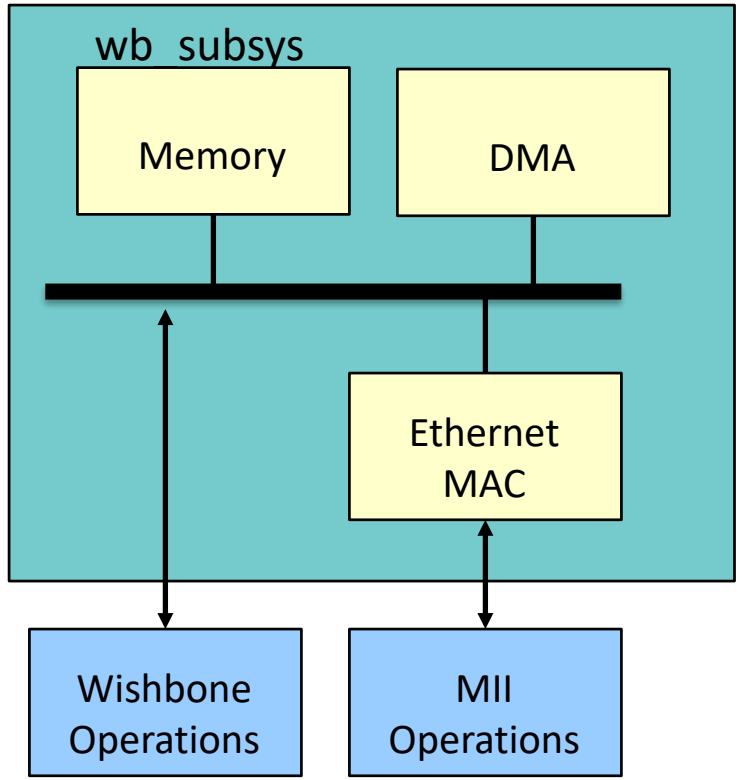
```
package package_name { body_item, ... }
```

Example:

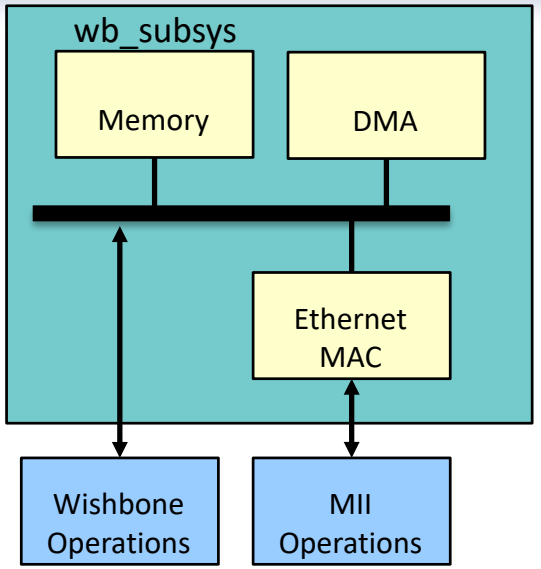
```
package data_flow_pkg {
  stream eth_packet {...}
  buffer mem_buff   {...}
}
```

data_flow_pkg

```
package data_flow_pkg {
  enum dir_e {Rx = 0, Tx};
  enum buff_type_e {MEM_BLOCK = 0, ETH_PKT}

  // Ethernet Packet definition
  stream eth_packet {
    rand dir_e dir;
    rand bit[15:0] payload_len;
  };
  // memory buffer definition
  buffer mem_buff {
    rand buff_type_e buff_type;
    rand bit[31:0] addr;
    rand bit[15:0] size;
  }
  // resources
  resource wb_bus {};
}
```



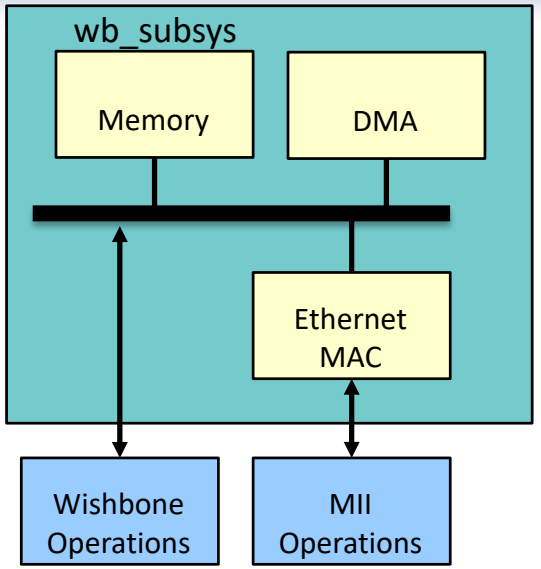
wb_ops_c




```
component wb_ops_c{
  import data_flow_pkg;  // import data_flow_pkg items

 // receive action
  action wb_receive_a {
    input mem_buff buff_in;
    exec body {
      pss_info("wb_receive_a","*** WB operations send action i *** \n\n");
    }
  }

  // send action
  action wb_send_a {
    output mem_buff buff_out;
    exec body {
      pss_info("wb_send_a","\n *** WB operatins send action *** \n\n");
    }
  }
}
```

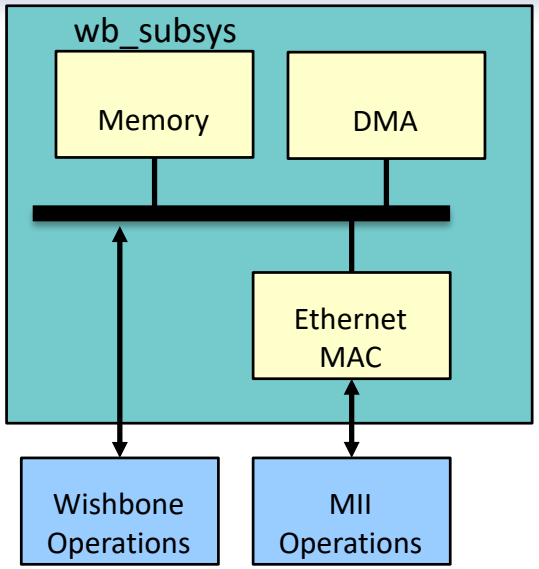
Simple printouts for test realization

mac_c



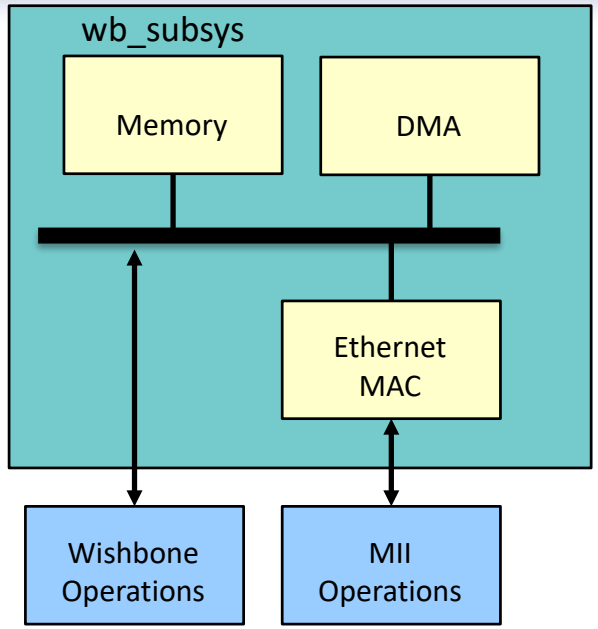

```
component mac_c {
  import data_flow_pkg::*;   // import data_flow_pkg items

  // action to receive an ethernet packet
  action rx_pkt_a {
    input  eth_packet pkt_in;   // input  eth packet
    output mem_buff   buff_out; // output wb mem buffer
    // Lock the Wishbone bus so transmit doesn't starve
    lock wb_bus wb_bus_l;
    // constrain eth_packet direction to receive only
    constraint pkt_dir_con {pkt_in.dir == Rx; }
    // constrain mem_buff type
    constraint buff_type_con {buff_out.buff_type == ETH_PKT; }
    exec body {
      pss_info("rx_pkt_a","\n *** MAC rx_pkt action *** \n\n");
    }
  }
```

mac_c (cont.)



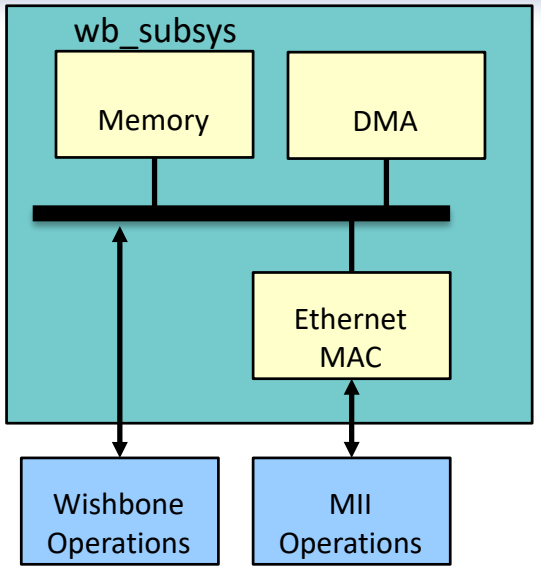

```
// action to transmit an ethernet packet
action tx_pkt_a {
  input  mem_buff   buff_in;  // input mem buffer
  output eth_packet pkt_out;   // output eth packet
  // Lock the Wishbone bus so transmit doesn't starve
  lock wb_bus wb_bus_l;
  // constrain eth_packet direction to send only
  constraint pkt_dir_con {pkt_out.dir == Tx; }
  // constrain mem_buff type
  constraint buff_dir_con {buff_in.buff_type == ETH_PKT; }
  exec body {
    pss_info("tx_pkt_a","\n *** MAC tx_pkt action *** \n\n");
  }  }
// action configure MAC
action config_mac_a {
  lock wb_bus wb_bus_l;      // lock wishbone bus
 exec body {
    pss_info("config_mac_a","\n *** MAC config action *** \n\n");
  }
}
}
```

mii_ops




```
component mii_ops_c {
  import data_flow_pkg;
  // send action
  action mii_send_a {
    output eth_packet pkt_out;  // output ethernet packet
    // constrain eth_packet direction to be to the MAC
    constraint pkt_dir_con{pkt_out.dir == Rx};
    exec body {
      pss_info("mii_send_a", "\n *** MII VIP send action *** \n\n");}
  }
  // receive action
 action mii_receive_a {
    input eth_packet pkt_in; // input ethernet packet
    // constrain eth_packet direction to be from the MAC (MAC transmit)
    constraint pkt_dir_con{ pkt_in.dir == Tx};      exec body {
       pss_info("mii_ receive _a","\n *** MII VIP receive action *** \n\n");
    }
  }
}
```

pss_top

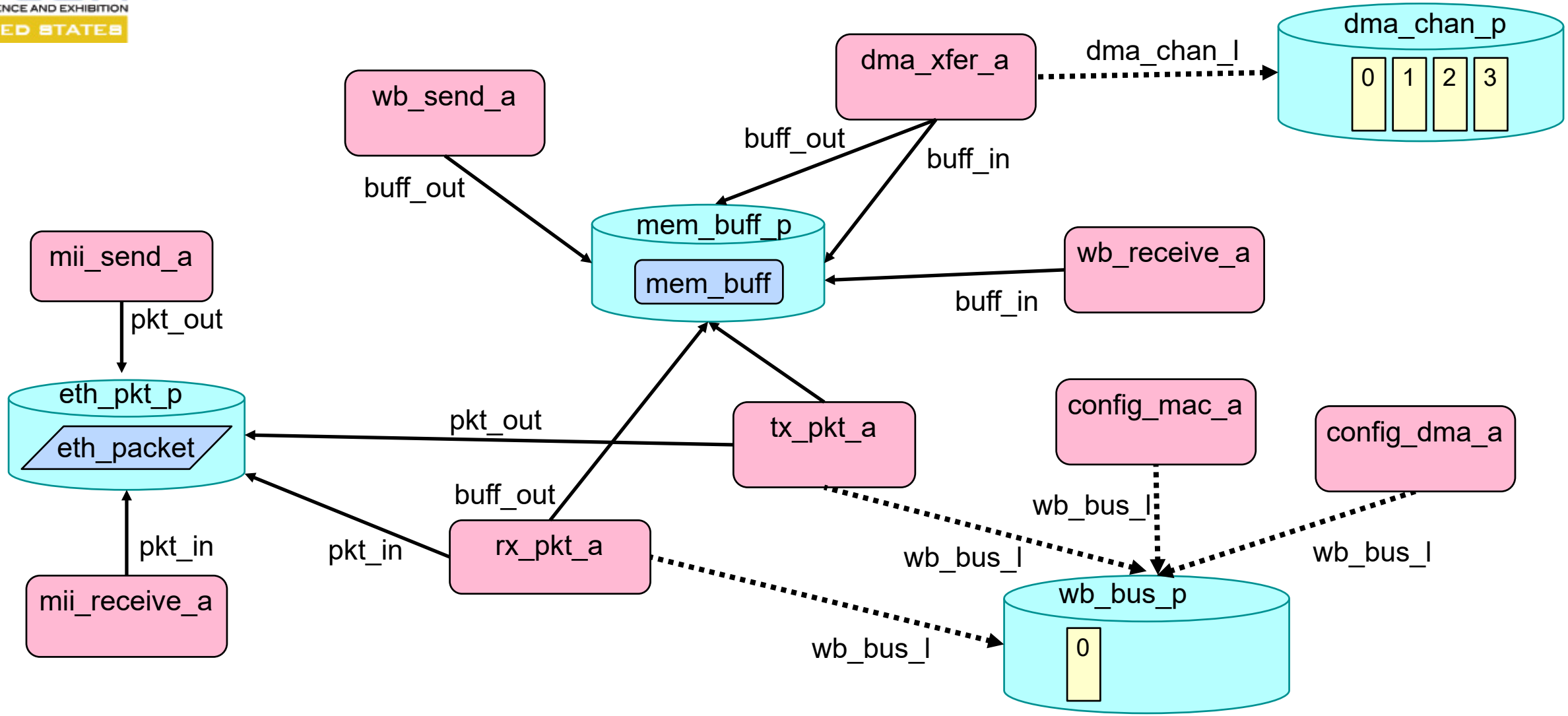



```
component pss_top {
  import data_flow_pkg::*;   // import data_flow_pkg items
  // component instantiations
  mac_c      mac;
  dma_c      dma;
  mii_vip_c  mii_vip;
  wb_vip_c   wb_vip;

  // pools
  pool eth_packet eth_pkt_pool;  // pool of ethernet packets
  pool mem_buff mem_buff_pool;   // pool of ethernet packets
  pool [1] wb_bus wb_bus_pool;

  // binds
  bind eth_pkt_pool   *;  // bind eth_pkt_pool  to *
  bind mem_buff_pool  *;  // bind mem_buff_pool to *
  bind wb_bus_pool    *;  // bind wb_bus_pool   to *
```

```
// entry action
  action entry_a {
    mac_c::rx_pkt_a rx_pkt;
    mac_c::tx_pkt_a tx_pkt;
    activity {
      schedule {
        rx_pkt;
        tx_pkt;
      }
    }
  }
}
```

"Compound" `action`: Schedules other actions in an `activity` block

wb_subsys Visualization



Test Realization

- We have scenarios that do correct actions
  - However if we asked Trek5 to generate tests from our scenarios the tests *would only do print statements as written*!
- Need to describe behavior in our action exec blocks that implement tests in the targeted test environment
  - The MAC `tx_pkt_a` action needs to "do" the transmit of a packet
- We want to take advantage of existing APIs, sequences etc. in our target test environment
  - Be it C code on a embedded processor or a sequence in a UVM testbench

PSS Test Realization

- PSS provides the ability to interact with foreign-language code
  - Help compute stimulus or expected values during stimulus generation
  - Calls to API or libraries that correspond to behavior in leaf-level actions
- PSS Procedural Interface (PI)
  - Defines mechanisms by which the PSS model can interact with other languages such as C/C++ and/or SystemVerilog
    - Import or export functions
    - Used to reference external foreign-language functions or classes
- *However*, PSS does not specify beyond "you can can import or export what you want"
  - Result is different "solutions" across different vendors for integration with their tool and "talking" with C or SystemVerilog

HSI

- Breker has defined a Hardware Software Interface (HSI) for use in exec blocks
  - Provides a standardized way of accessing registers, memories and VIP
  - Provides a translation layer that hides underlying details
- For accessing registers and memories
  - Defines a "register model" very similar to the UVM register model
    - Similar API methods for writing, reading, setting, updating registers and memories
    - Provides a translation layer for register and memory accesses
- For communication with VIP etc.
  - Defines TLM style transactions and ports with TLM methods (get, put etc.)
- HSI code gets realized in target language (C , SV etc.)

HSI Example – Uart Register Block

```
class uart_block : public hsi::reg_block {
public:
  uart_block(
    const pss::scope& name, hsi::reg_addr base, const std::string& tb_path)
    : hsi::reg_block (this),
      map("map", base),  cfg_port("cfg_port", tb_path + "_cfg"),
      drv_port("drv_port", tb_path + "_drv"), chk_port("chk_port",
      tb_path + "_chk")
  {
    map.add_reg ( UART_RX,          0x00 ); // R
    ...
    map.add_reg ( UART_LSR,         0x05 ); // R
    ...
   }
  reg_uart_DATA    UART_RX          { "UART_RX"          };
  ...
  reg_uart_LSR     UART_LSR         { "UART_LSR"         };
  hsi::reg_map map;
  hsi::put_port   <uart_cfg_tlm>   cfg_port;
  hsi::put_port   <uart_drv_tlm>   drv_port;
  hsi::check_port <uart_chk_tlm>   chk_port;
};
```

adding and mapping registers

TLM Ports

code in examples/wb_subsys_v2a

HSI Example – Uart Configuration Action

```
class UartCfg : public pss::action {
 ...
  void config(uart_block& blk) {
    hsi::status_e status;
    ...
    // config uart
    blk.UART_LCR.DIVISOR_ACCESS.set(0);
    blk.UART_LCR.PARITY_ENABLE.set(1);
    ...
    blk.UART_LCR.update(status);

    // config uart VIP to match DUT configuration
    uart_cfg_tlm cfg_tlm {"cfg_tlm"};
    cfg_tlm.parity_enable.set ( blk.UART_LCR.PARITY_ENABLE.get() );
    cfg_tlm.parity_even.set   ( blk.UART_LCR.PARITY_EVEN.get()   );
    cfg_tlm.char_size.set     ( blk.UART_LCR.CHAR_SIZE.get()     );
    blk.cfg_port.put(cfg_tlm); }

  void body() {
    config(*sys.uarts.at(0)); // config uart0
    config(*sys.uarts.at(1)); // config uart1
  }
```

Writing configuration registers in the UART

Create a TLM transaction

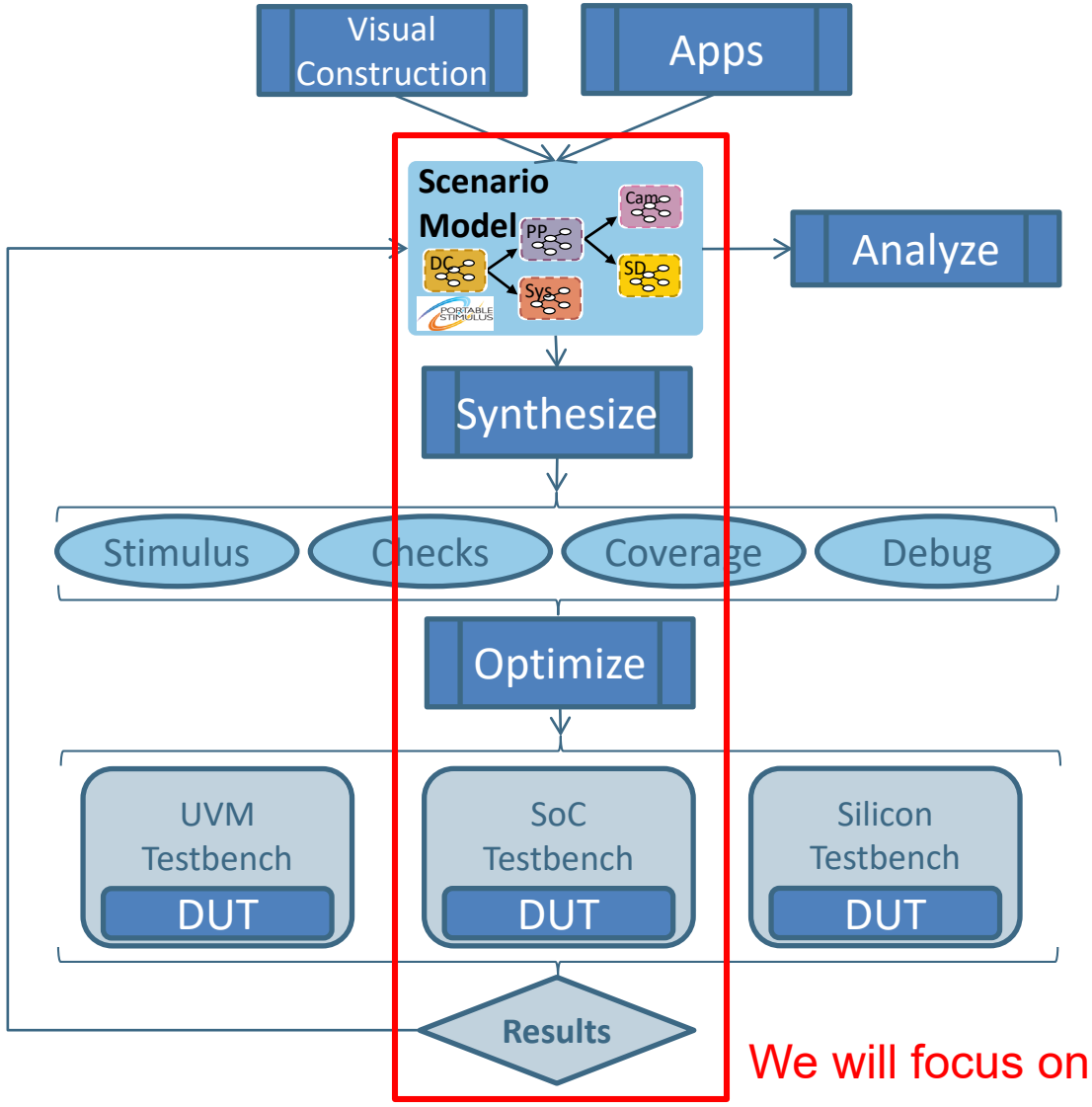
Send transaction to VIP

Do the UART configuration

code in examples/wb_subsys_v2a

Executable Specification Test Case Synthesis

Abstract, scenario test-case synthesis, for all stages of the verification process, from a single, comprehensible, executable intent specification



Advanced modeling, Apps for common challenges

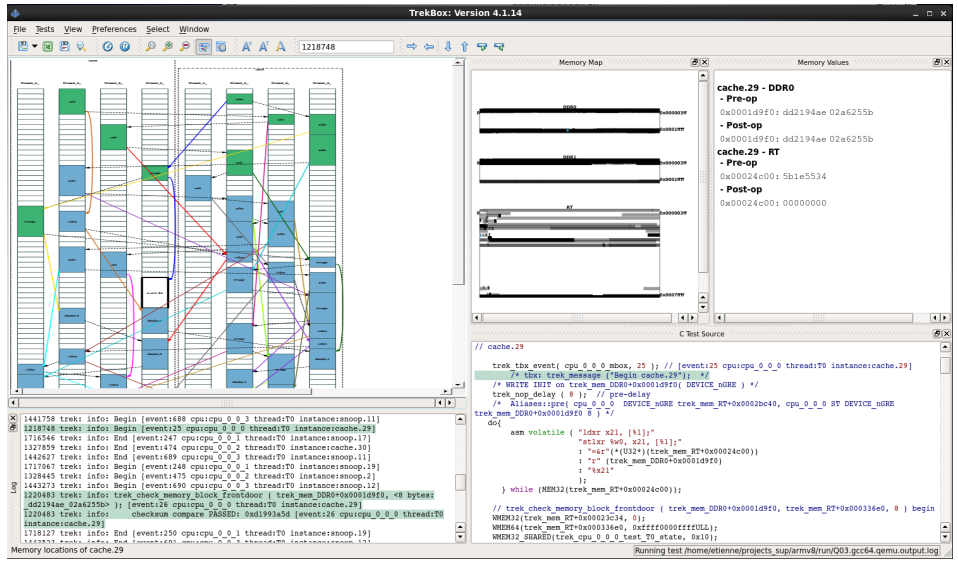
Verification Space Analysis and Optimized Test Generation

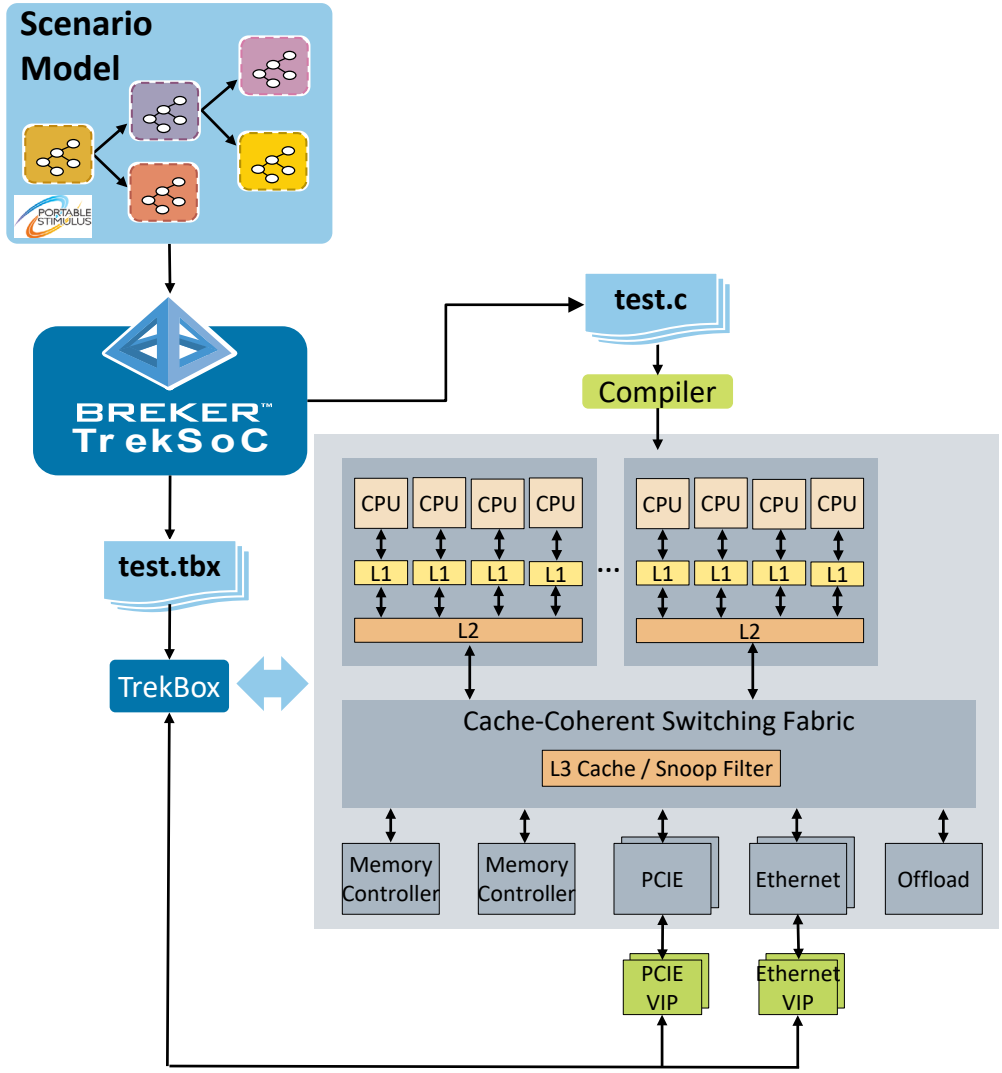
Platform Specific Test Optimization

We will focus on the SDV flow

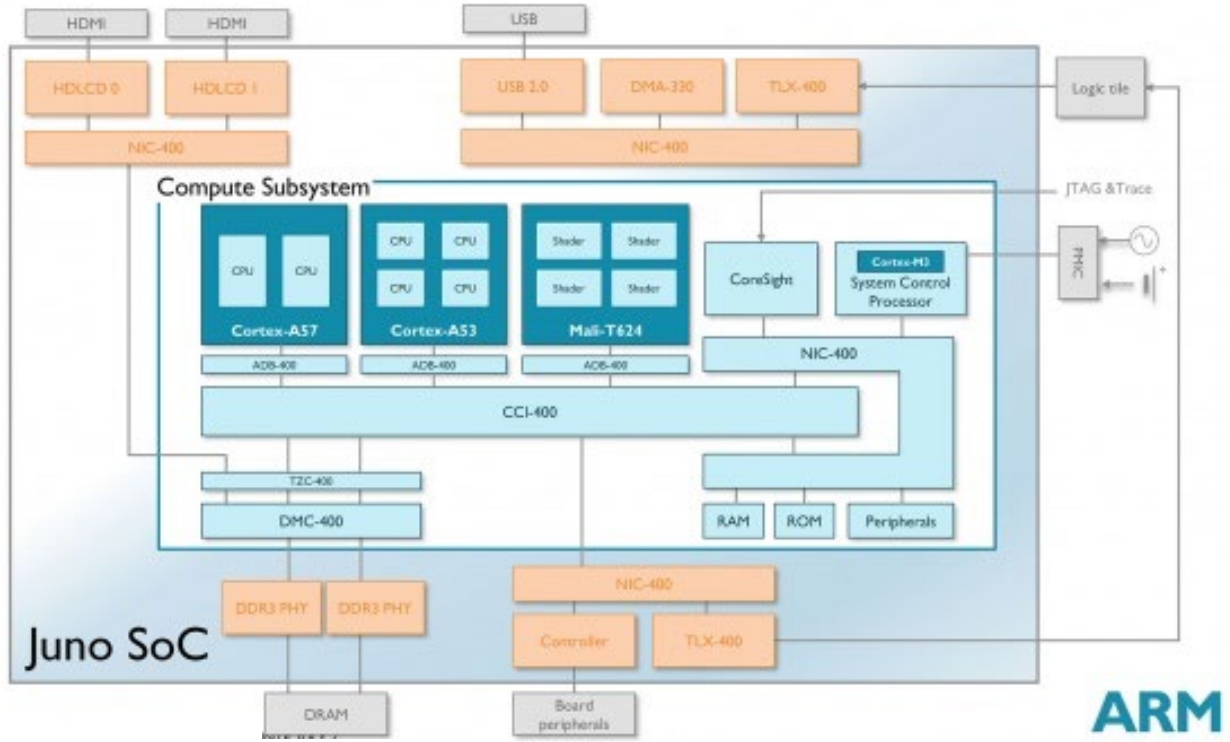
SDV Portable Stimulus Deployment Example

- Broad, comprehensive test sets synthesized to exercise corner-cases, hard to write by hand
- Memory management, hardware software interface, “trickboxing” for full automation
- Debug visualization of concurrent, synchronized transaction and SW tests with backdoor access





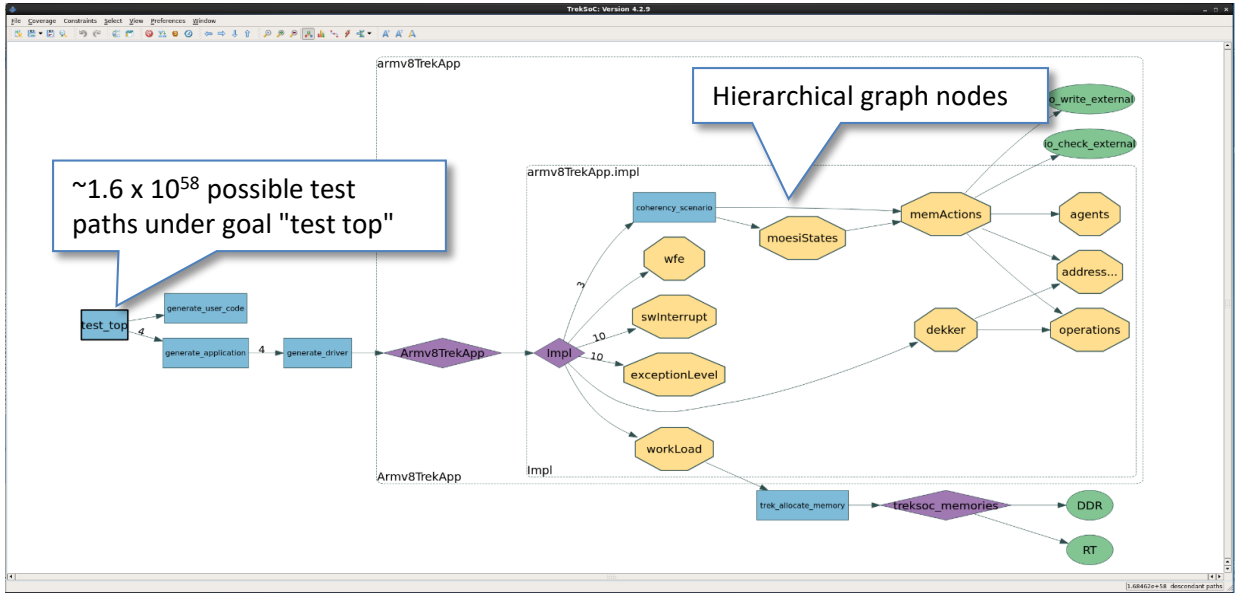
ARM Platform Verification Issues



This infrastructure verification process needs a large number of tests to check, for example:

- Cache coherency, Stress testing of sub-system components, SoC functional testing, etc.

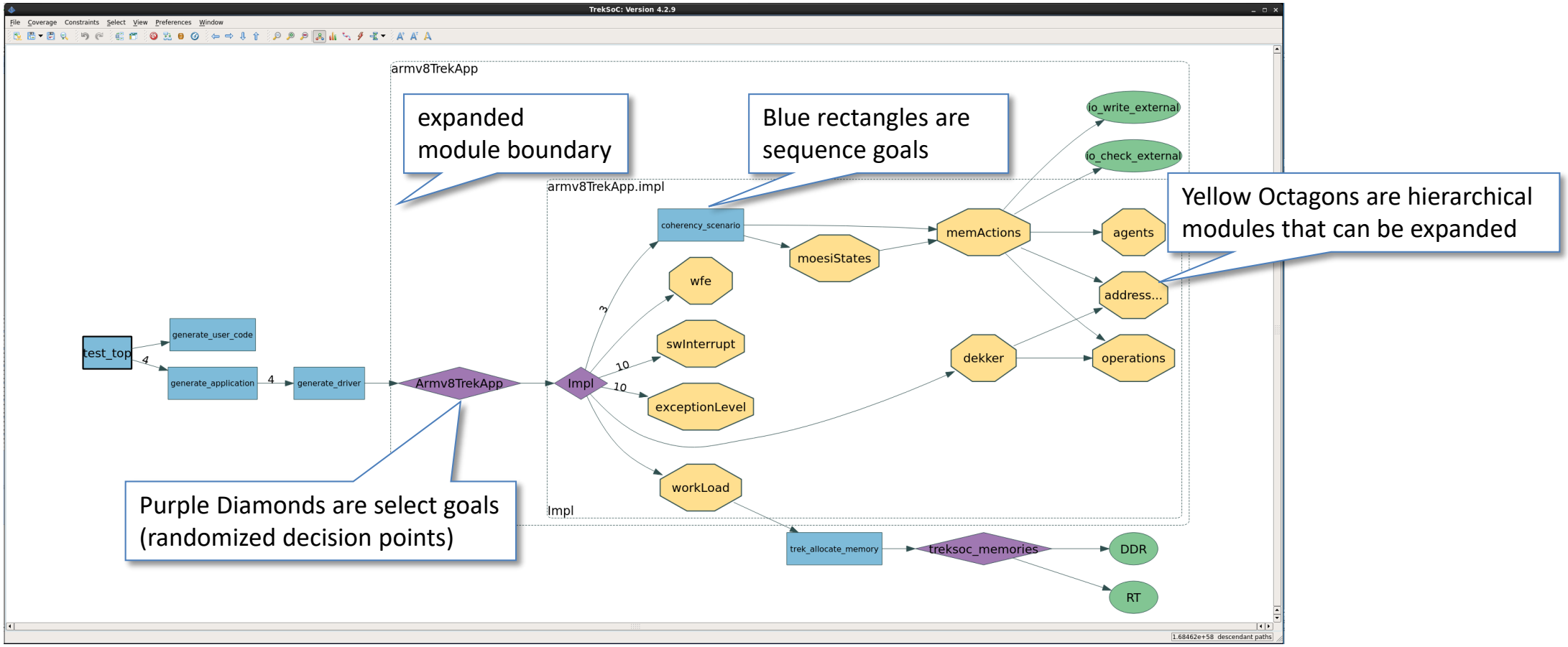
ARMv8 Application
Easy verification of ARM installations



Automated test generation for a broad range of ARMv8 integration issues, including in-depth SoC cache coherency.

- Auto-generation of broad, inclusive test sets, otherwise requiring man-months of manual authoring
- Find and wring out complex, SoC corner cases hard to envisage manually
- Complete ARMv8 integration verification for SoC simulation, emulation and post silicon

Top Level ARMv8 Graph Structure



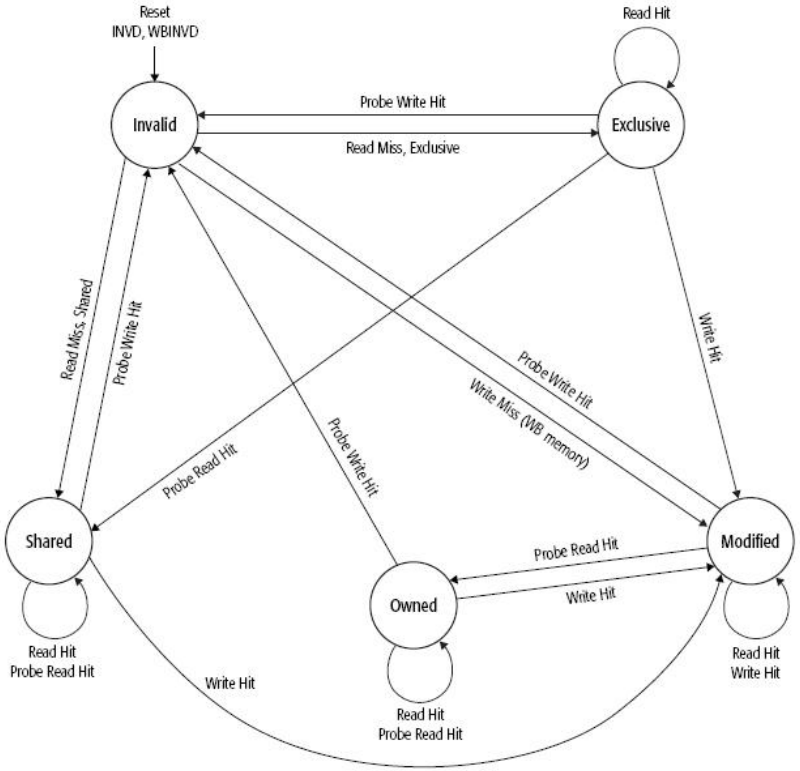
Tool Traverses Scenario Model graph to generate tests

ARMv8 Verification Metrics

- Cache State Transitions
- Cache Line Sharing Cases
- Snoop / Probe Sources
- Load/Store Operation Size
- Load/Store Sources
- False Sharing Cases
- Crossing Cache Line Boundaries
- Capacity Eviction Cases
- Multiple Memory Regions
- Memory Ordering Tests
- Concurrent Scenarios
- Interrupt/Exception Sources

Cache State Transitions

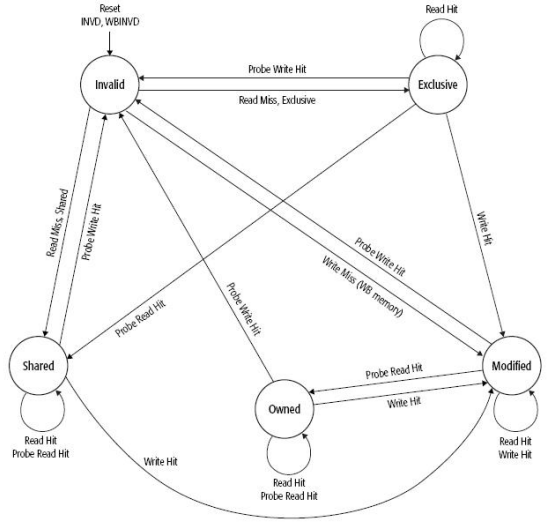
- There can only be up to five states in global context
- Need to follow a specific sequence of **transitions** to reach each **state**



source: http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2011/ch8_cl

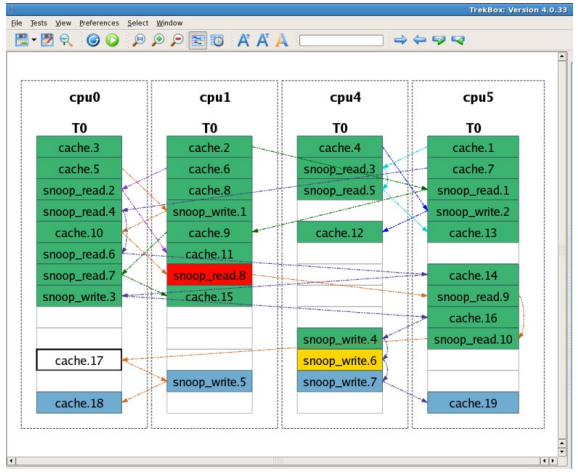
Cache State Transitions

- "One Address, Many Data"
- Start with end state, work backwards to find transition scenario

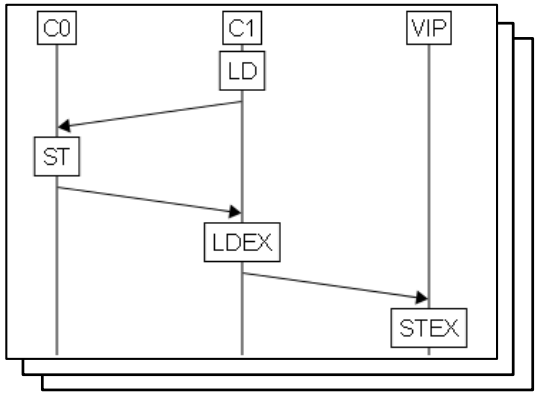


N Transition Sequences

Concurrent Scenario Test Case

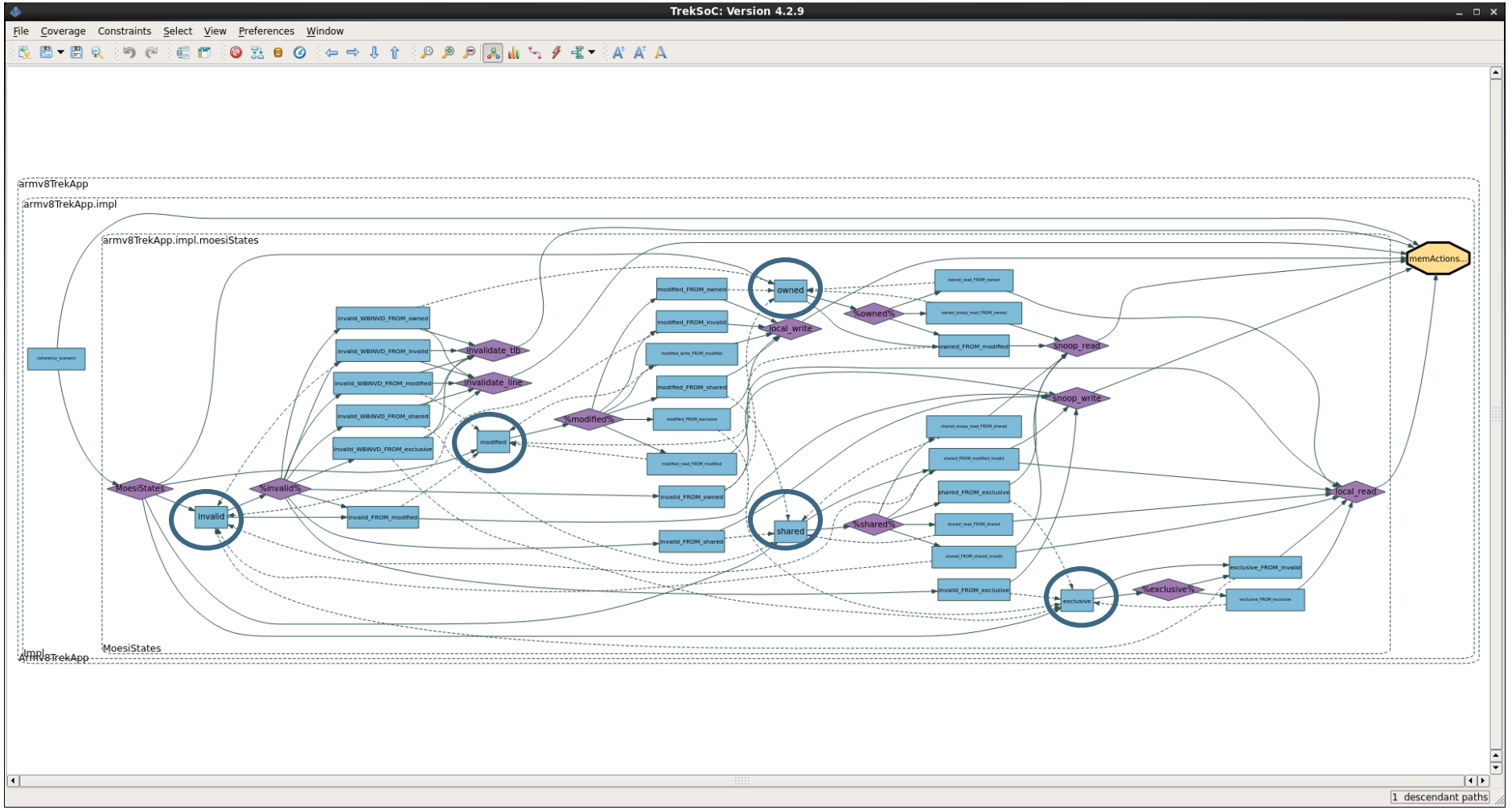


N Transition Scenarios



Schedule Memory
Interleave & Pack
Resolve Dependencies

Cache State Transition Graph



Cache Line Sharing Cases

- Need to consider all possible cache line sharing cases across caches
  - How many caches are sharing the cache line
  - Which caches are involved
  - Is the shared line clean or modified

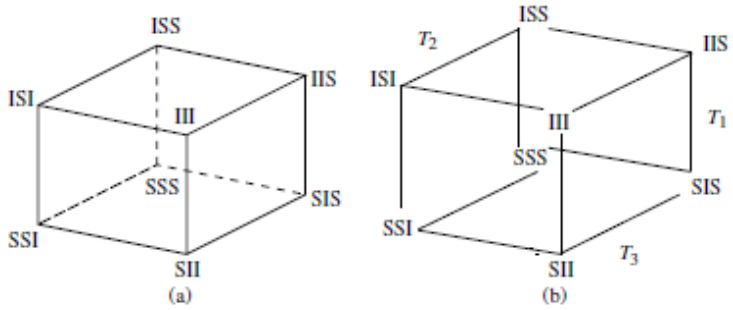


Fig. 2. (a) State space of SI protocol with 3 cores. Each global state is presented with 3 letters, e.g., IIS means core 2, core 1, and core 0 are in states I, I, and S, respectively. (b) Viewed as a composition of 3 isomorphic trees.

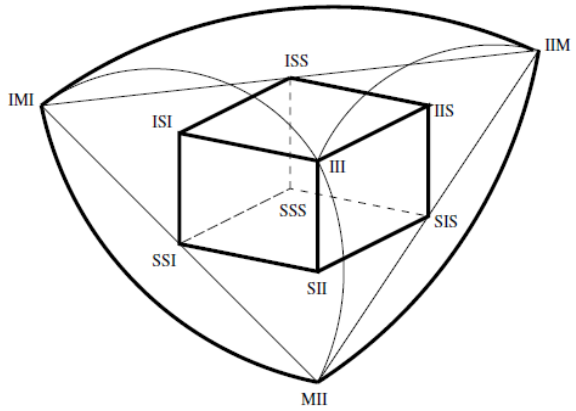
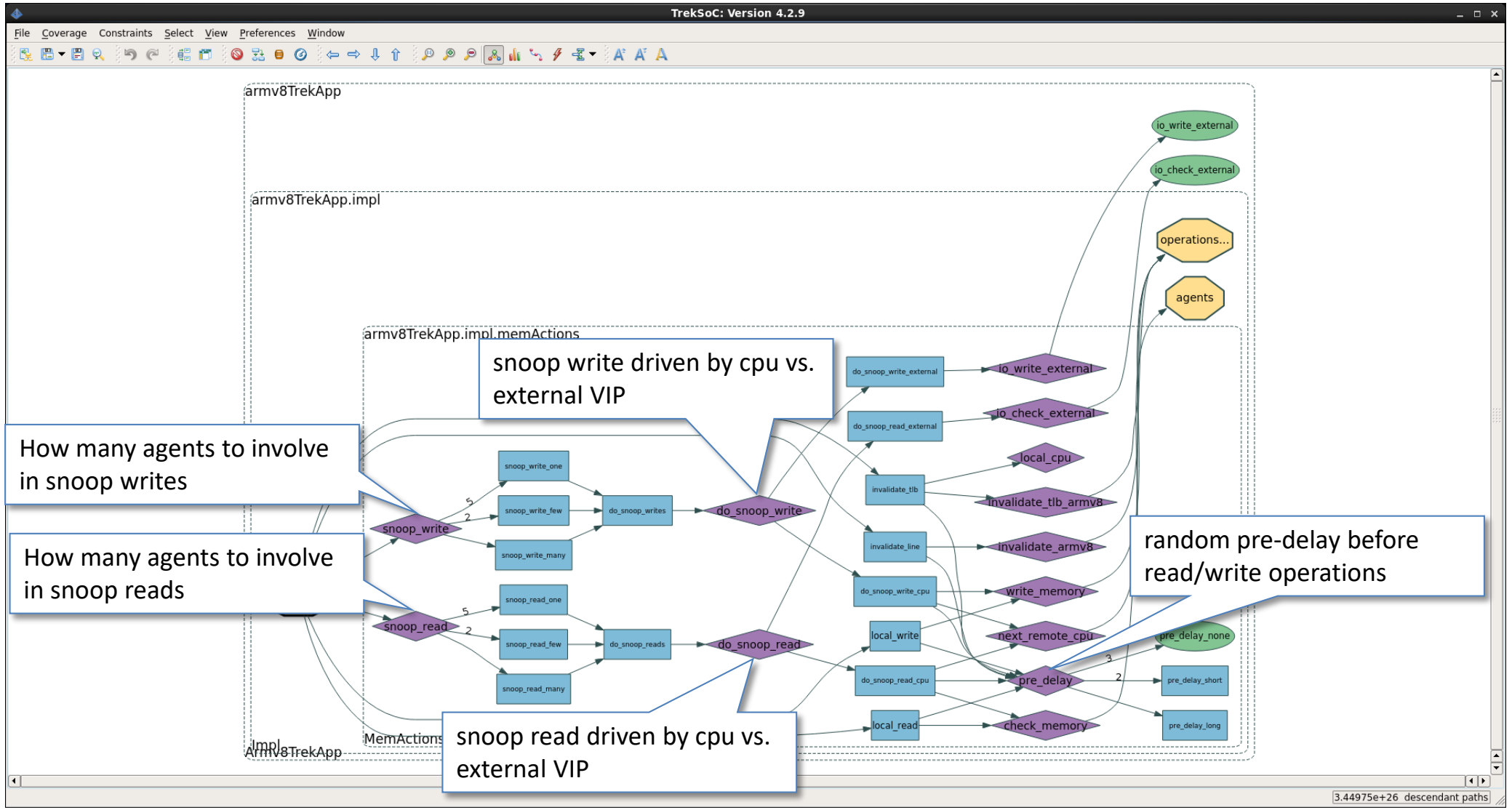


Fig. 3. State space of MSI protocol with 3 cores. For the clarity of presentation, the transitions to global modified states (IIM, IMI, MII) are omitted, if the transition in the opposite direction does not exist.

source: Qin et al., http://www.cise.ufl.edu/tr/DOC/REP-2012-537.pdf

Cache Line Sharing Cases

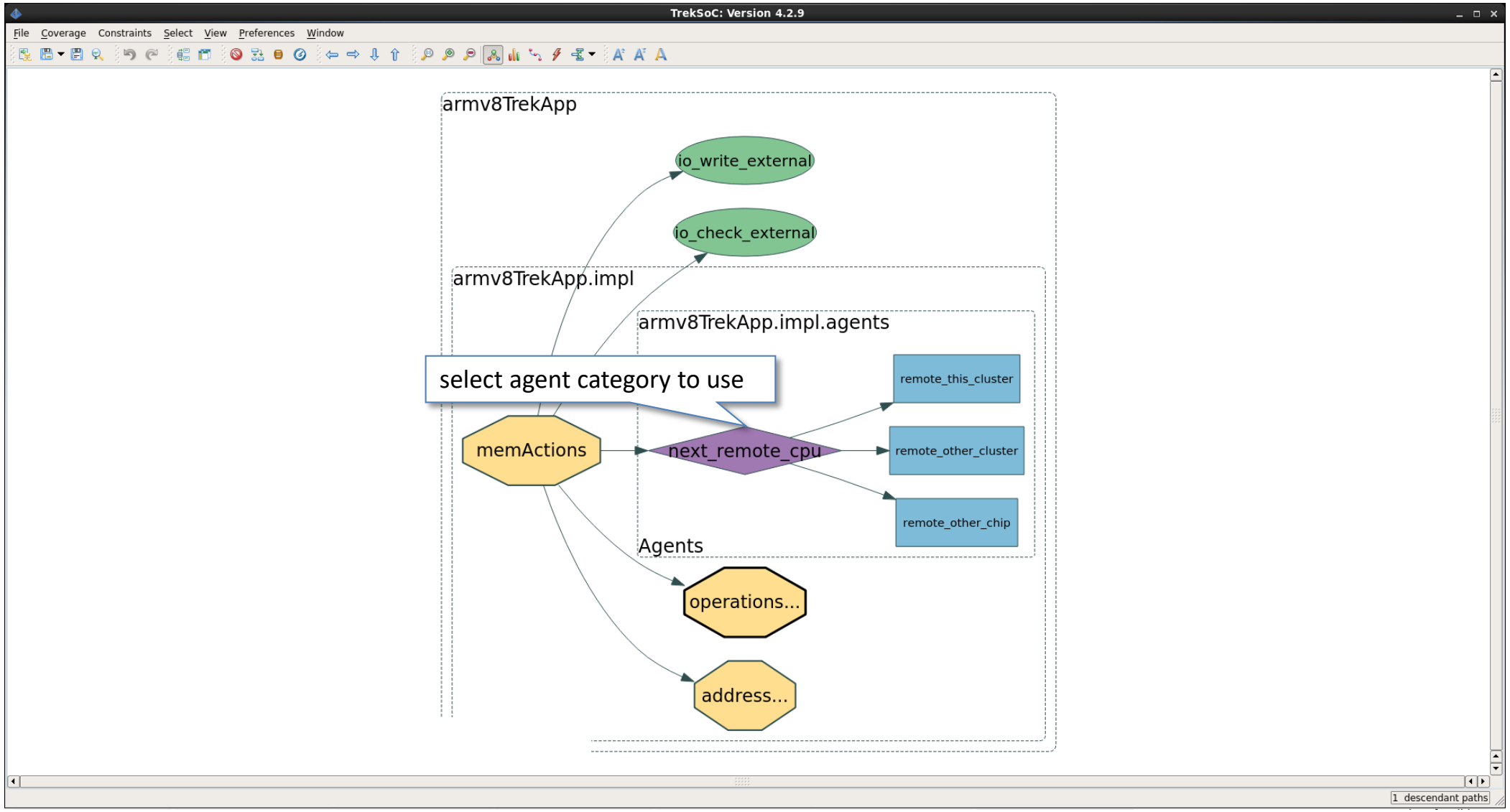


Snoop/Probe Sources

- Snoops/Probes query a cache to see if it contains a cache line
  - Cache may respond by writing back / returning dirty data

- Need to consider multiple Snoop / Probe sources
  - Another core on the same cluster
  - A core from a another cluster
  - A core from another chip
  - A coherency master (e.g. PCIe)

Snoop/Probe Sources

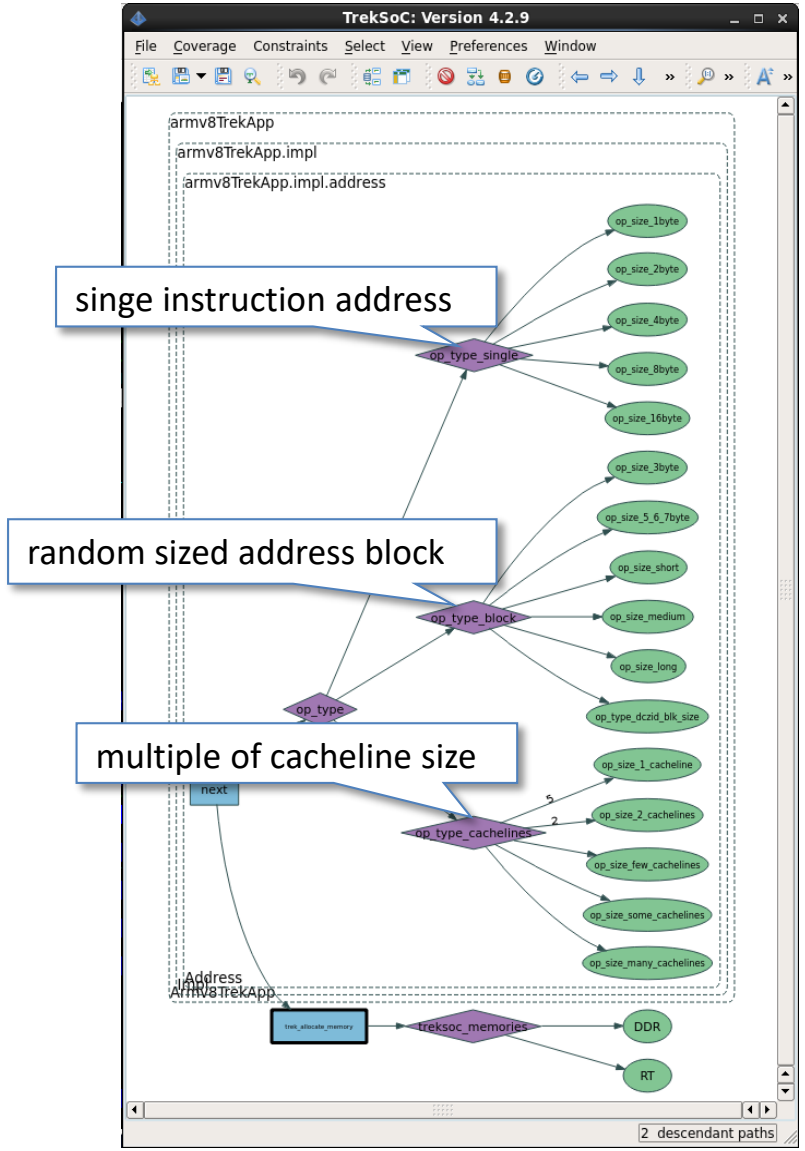


Load/Store Operation Sizes

- Must consider 1 Byte, 2 Byte, 4 Byte, 8 Byte operations for Loads and Stores to caches
  - Can do many small operations concurrently
  - Opportunities for false sharing (more on this later)

- Must also consider block operations
  - Do a sequential Load or Store operations to a block of addresses (e.g. 3233 bytes)
  - Causes fetch buffers, write buffers, branch prediction etc to fill up
  - Different type of testing than with single, small operations
  - Some operations require blocks that are multiples of cache-line size

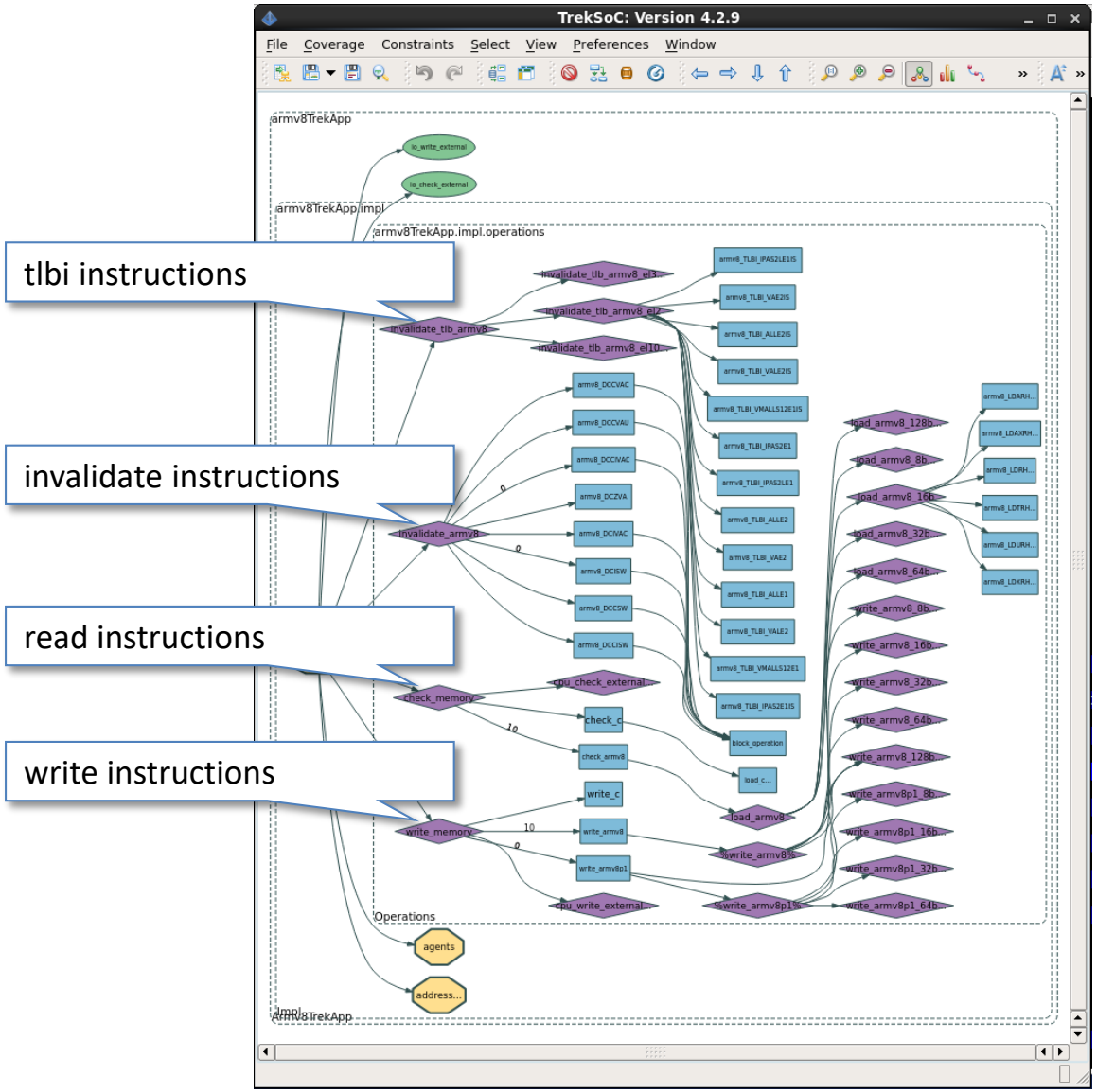
Load/Store Operation Sizes



ARMv8 Load/Store Instructions

- All variants of Load/Store operations including
  - Acquire/Release
  - Exclusive
  - Pair Operations (16 byte)
  - All valid sizes of the above

ARMv8 Load/Store Instructions



Multiple Memory Regions

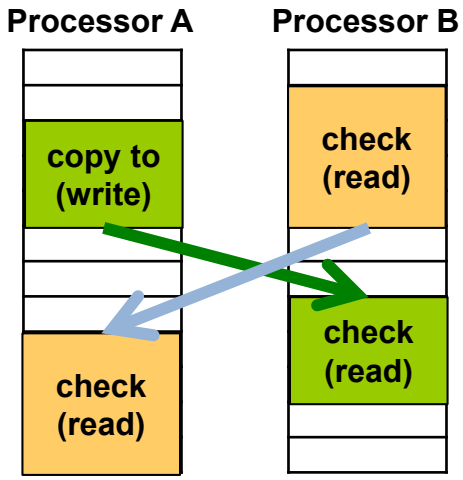
- A memory region is a range of addresses at a specific location in the memory map
- Randomize memory addresses across multiple memory regions with different properties
  - Different cache-ability properties
  - Different memory controllers
  - Different physical memory types
- Memory regions configured in configure/platform.trekcfg

Memory Ordering Tests

- CPU0
  - writes data A
  - (memory barrier)
  - write flag B
- CPU1
  - wait for flag B
  - read data A
- CPU1 *must* get data value A from CPU0

Memory Ordering Tests

- Every dependency that crosses processors tests memory ordering
  - Producer writes data
  - Producer updates state (with memory barrier)
  - Consumer waiting for state
  - Consume uses data
- Tested on every producer/consumer dependency
- See Test Map view



Dekker Algorithm

- Assume initial state A=0 , B=0

- The Dekker Algorithm States

```
core 0: ST A, 1; LD B
core 1: ST B, 1; LD A
error iff ( A == 0 && B == 0 )
```

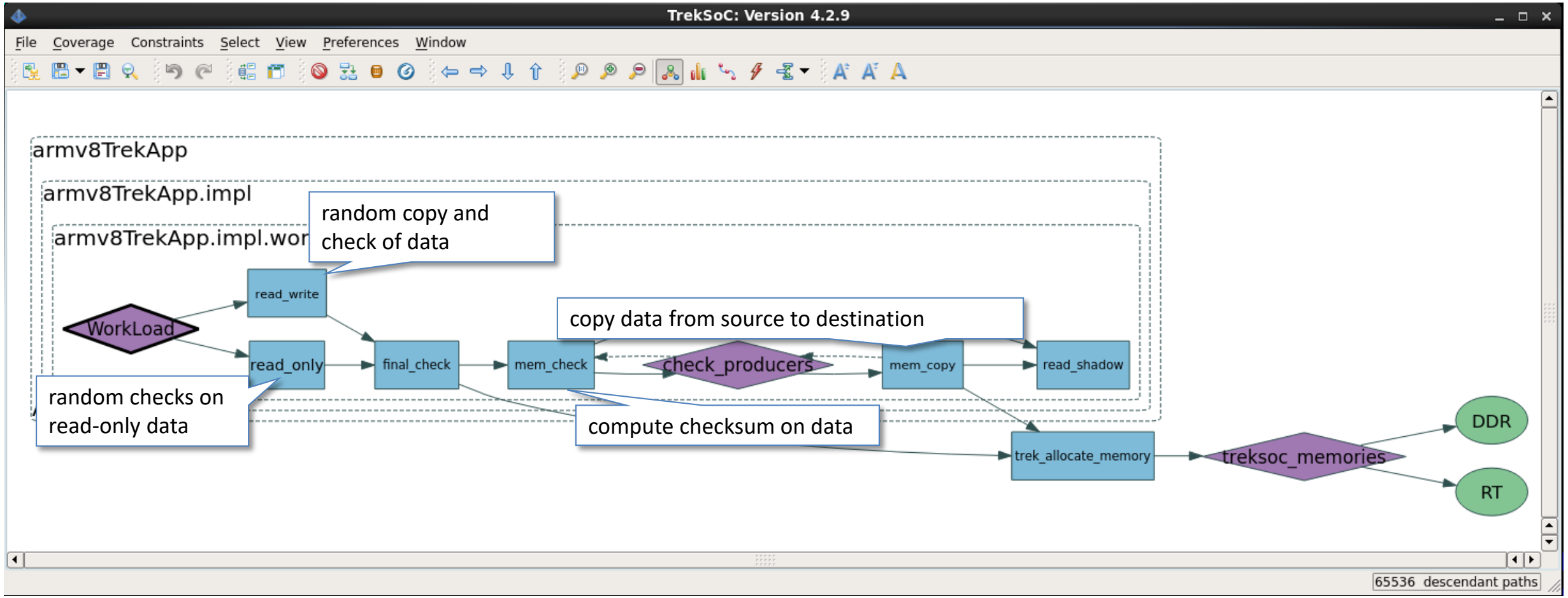
- This is a test for a weakly ordered memory system
  - Such a system must preserve the property that a LD may not reorder ahead of a previous ST from the same agent

Dekker Scaled to Multiple Processors

Core 0	Core 1	Core 2	Core 3
ST A	ST B	ST C	ST D
LD B	LD C	LD D	LD A

- Error if all loads see initial value
- Dekker randomized for all memories, operation sizes, load/store sources

Load/Store Sources



Summary

- PSS provides a powerful method to raise the abstraction for multiple verification flows
- PSS combined with the right tooling allows for powerful verification solutions with the minimal of user coding effort
- ARMv8 integrations are one area where PSS can discover a range of issues