2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Practical Tips for Adopting PSS

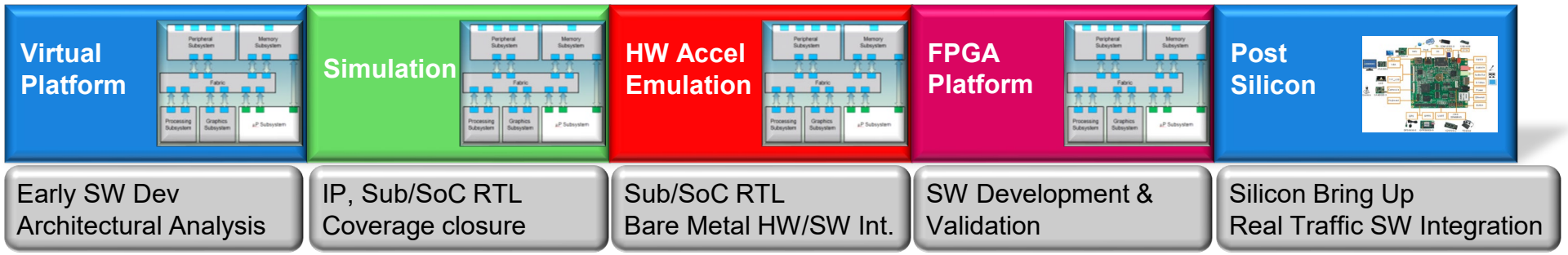
Tom Fitzpatrick
Portable Stimulus Working Group

Agenda

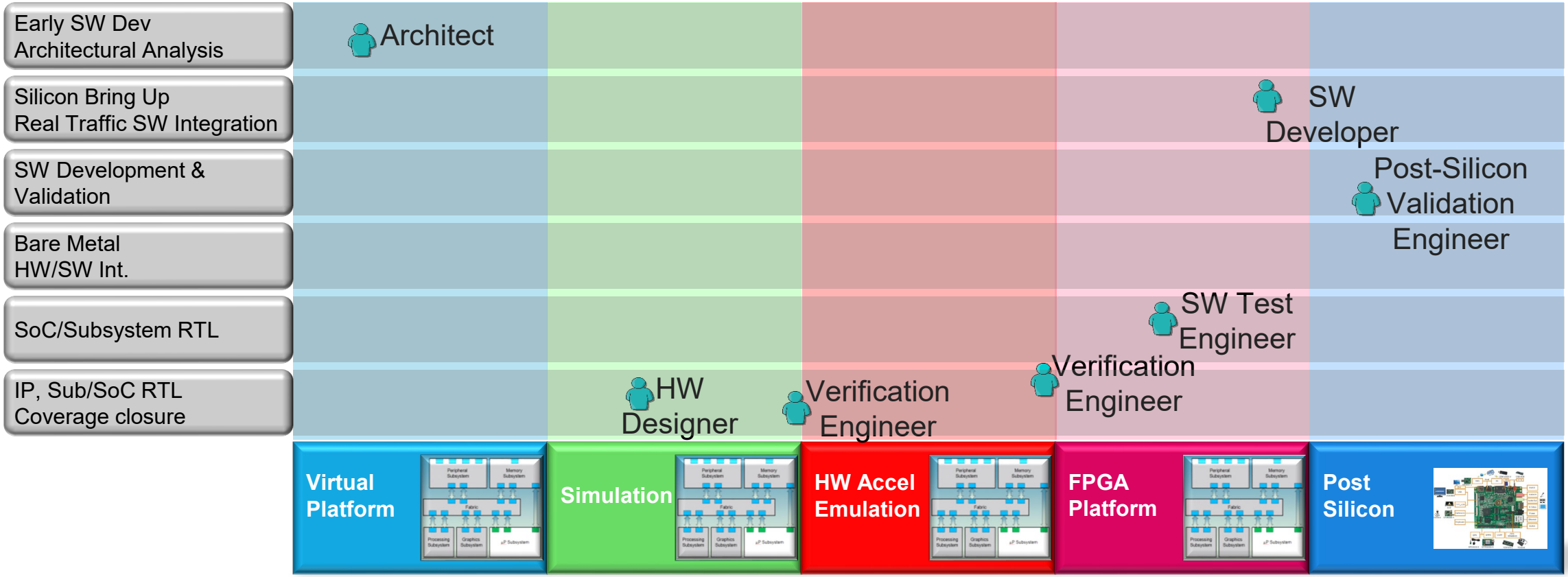
- Why PSS?
- PSS Layering
- Modeling and Automating Device Configuration
- Realizing PSS Scenarios in UVM and C

The Need for Portable Stimulus

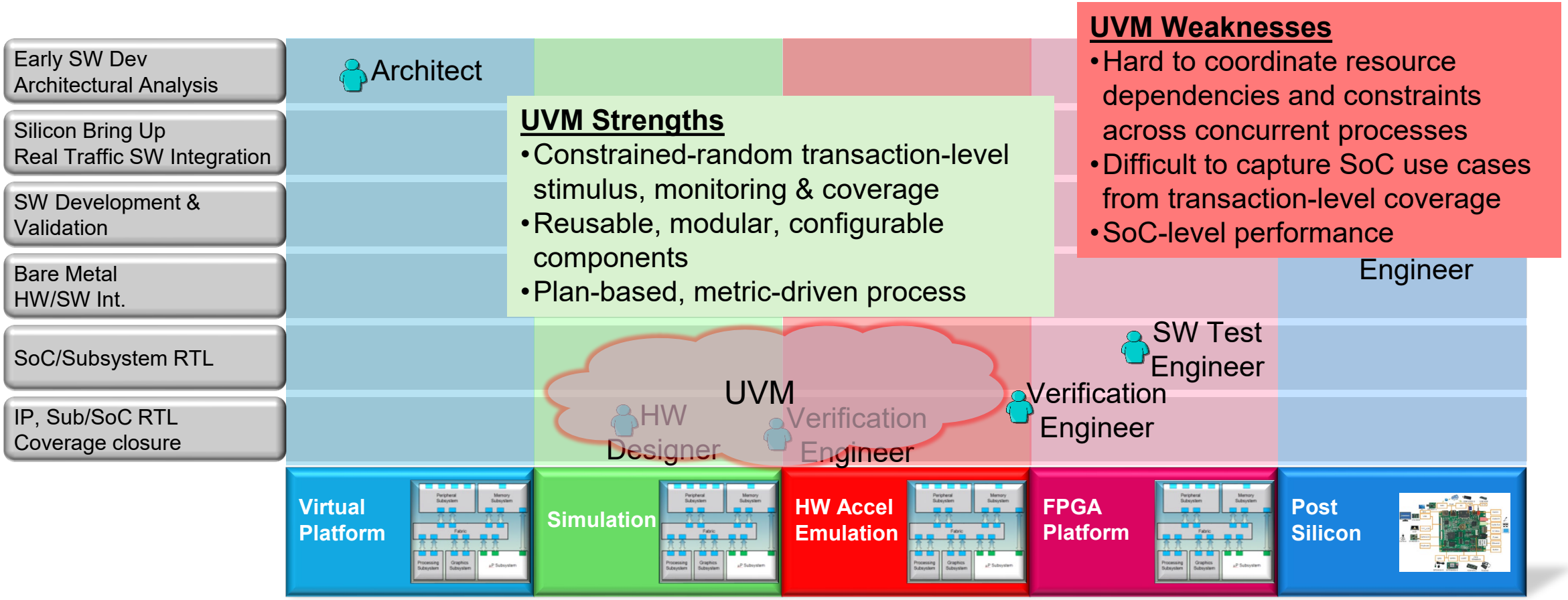
- ✖ ***Many specialized engineering resources required***
- ✖ ***Significant development effort for each environment***
- ✖ ***Limited sharing of models between environments***
- ✖ ***Difficult to reuse tests across environments***
- ✖ ***A lot of effort to migrate between environments/projects***



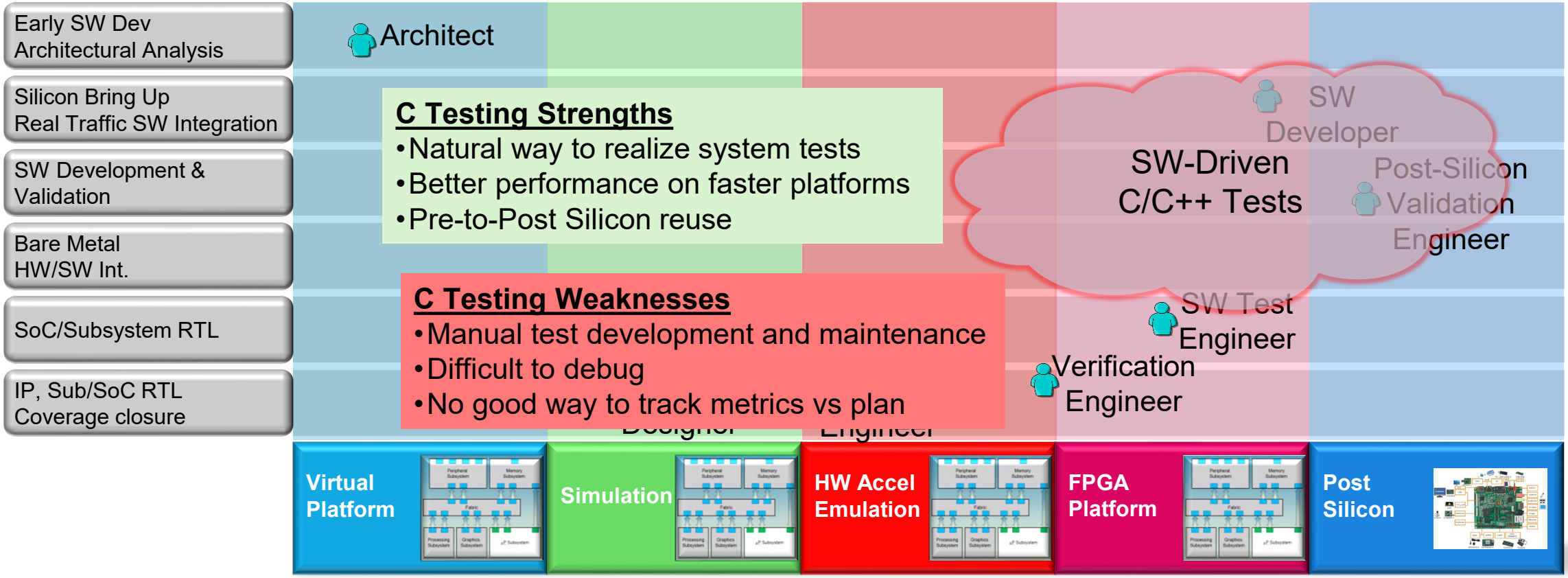
The Need for Portable Stimulus



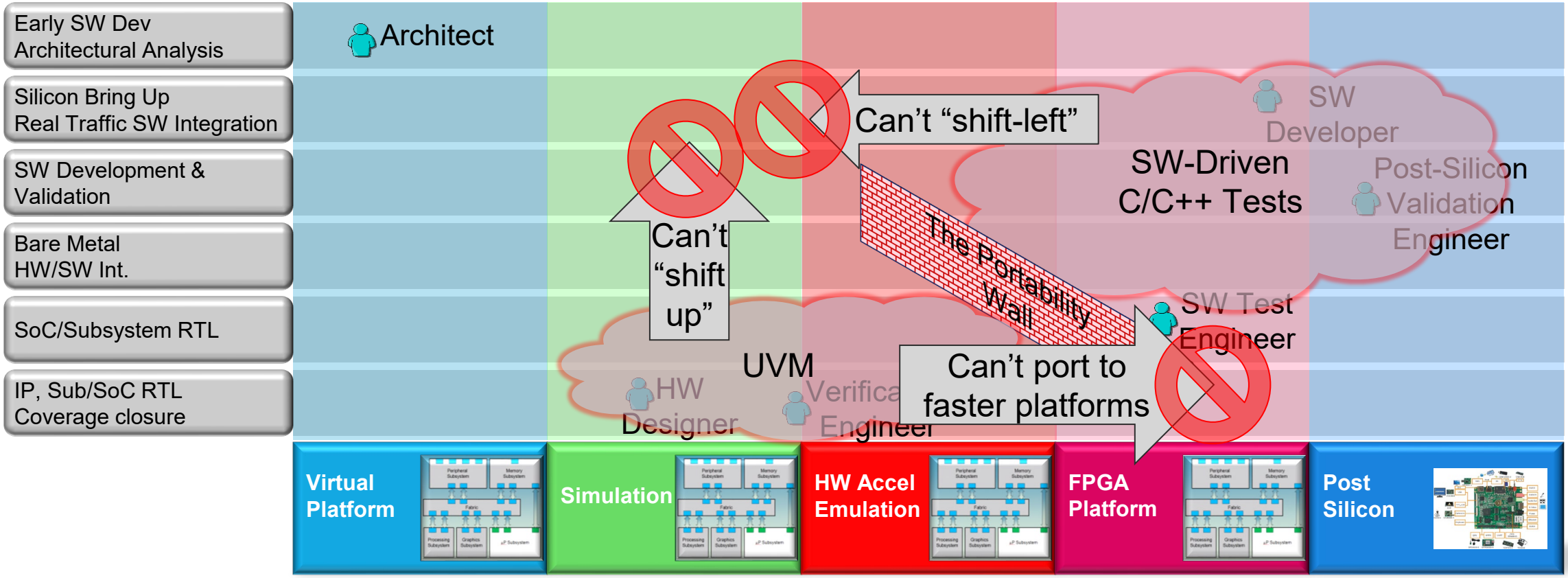
The Need for Portable Stimulus



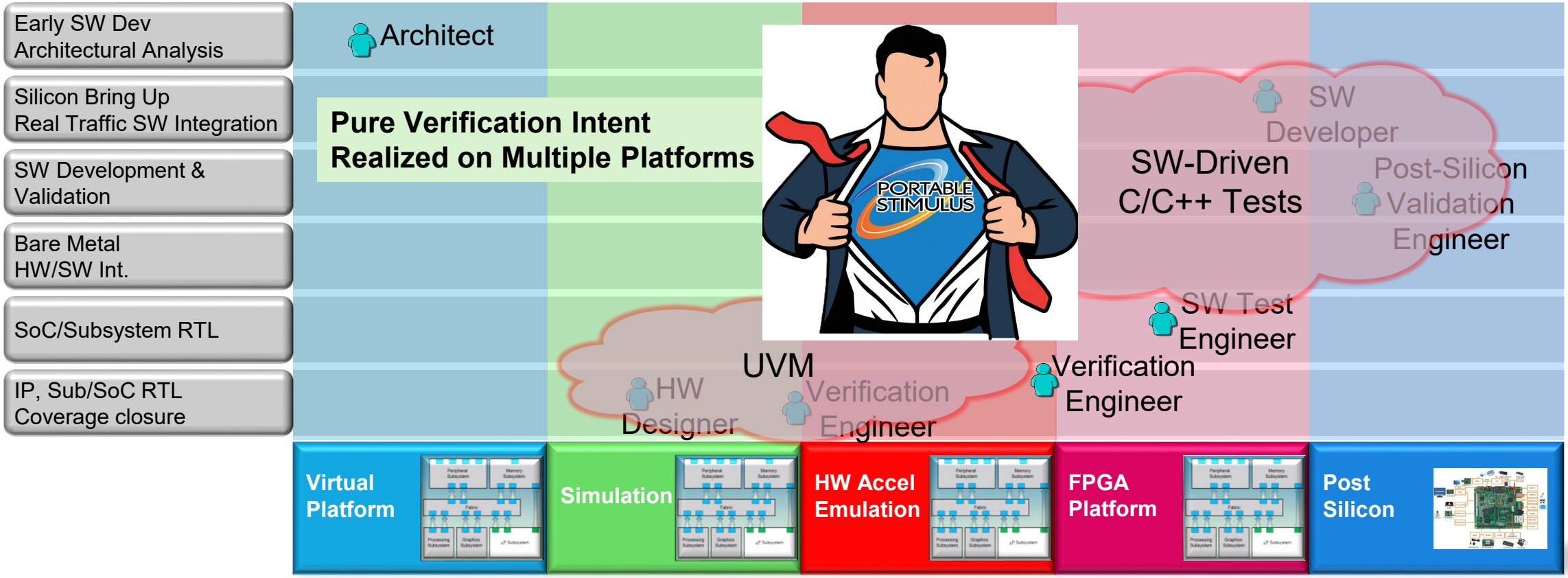
The Need for Portable Stimulus



The Need for Portable Stimulus



The Need for Portable Stimulus

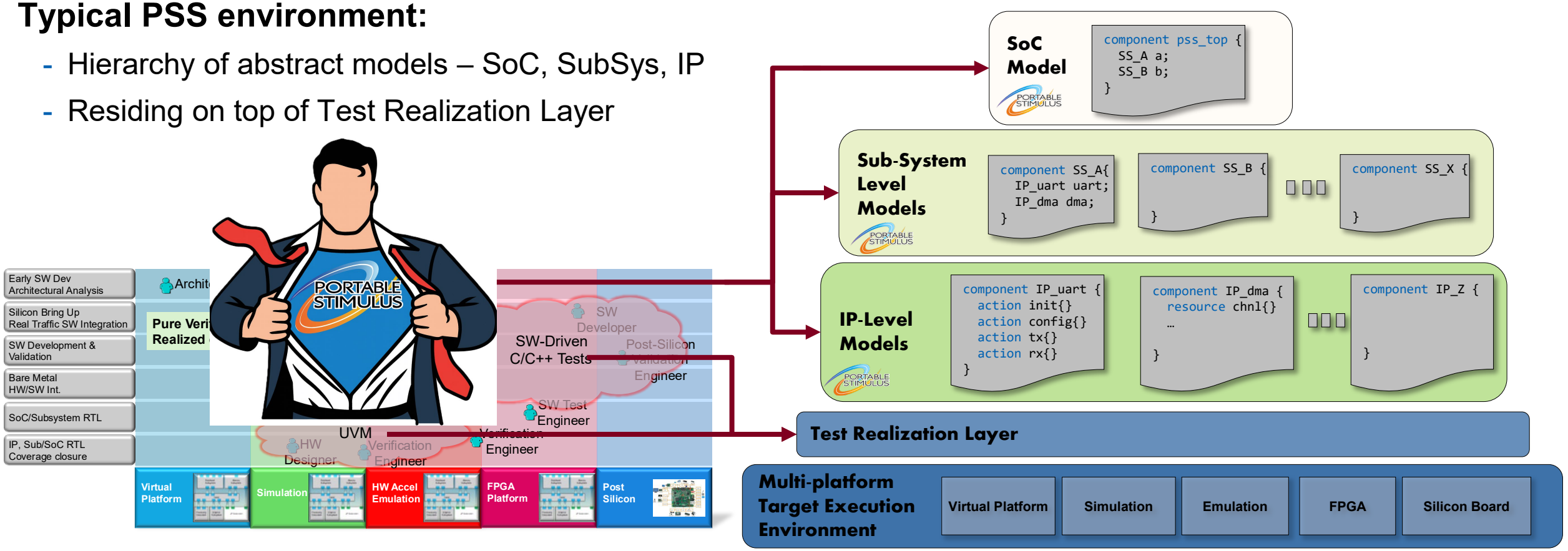


The PSS View of a Modern SoC Design:

From Vision to Deployment and Production

Typical PSS environment:

- Hierarchy of abstract models – SoC, SubSys, IP
- Residing on top of Test Realization Layer



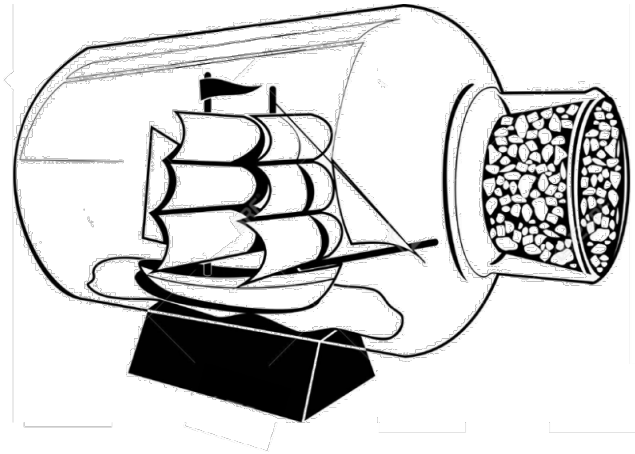
2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

PSS is Like an Onion(/Ogre): Pss Has Layers

What is a Portable Stimulus Model?

The Abstract Model

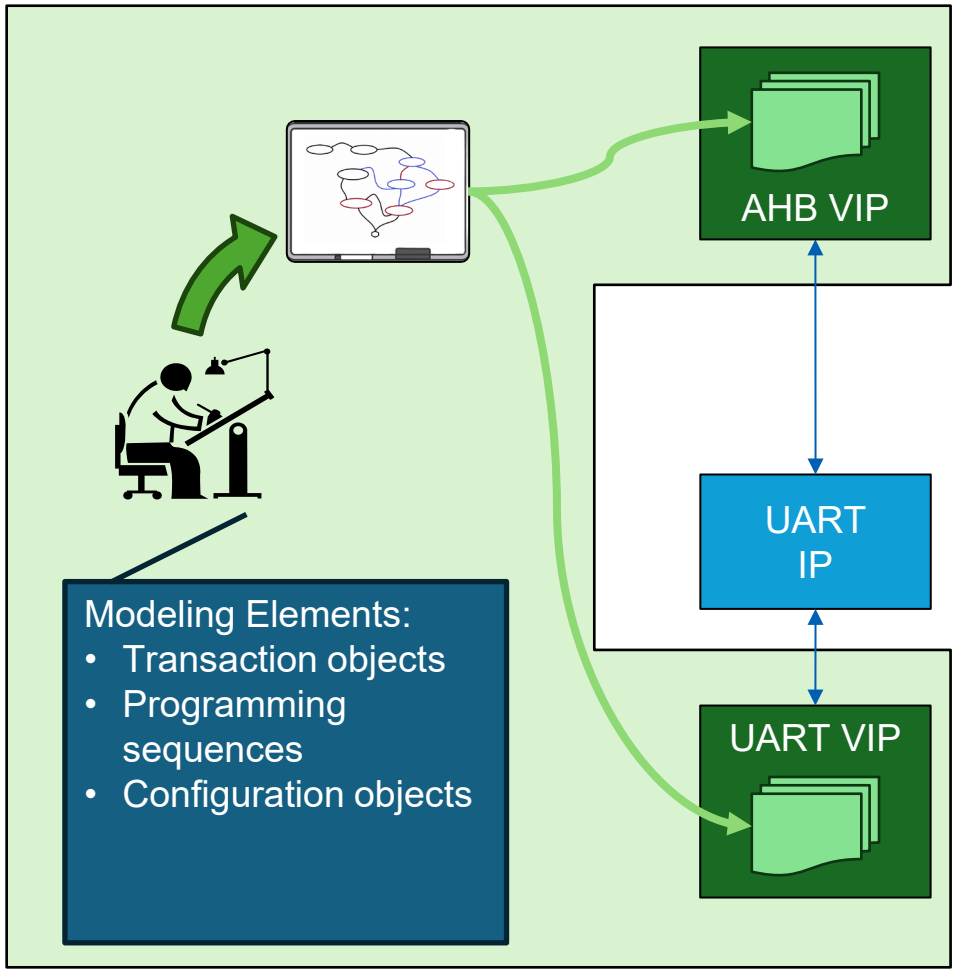
- *What* does it do



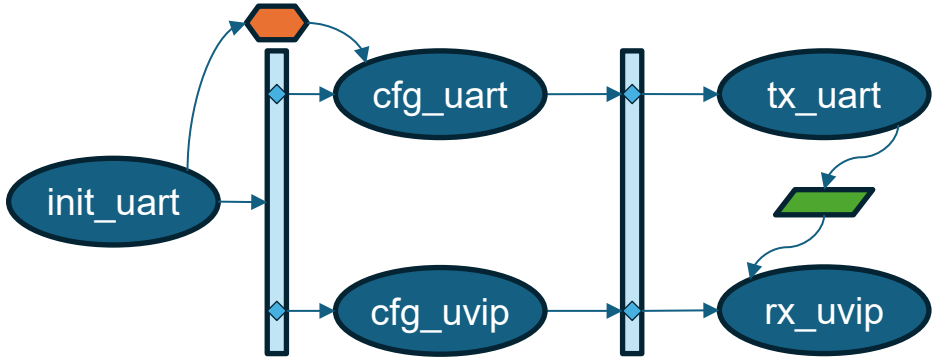
The Realization Layer

- *How* does it do what it does

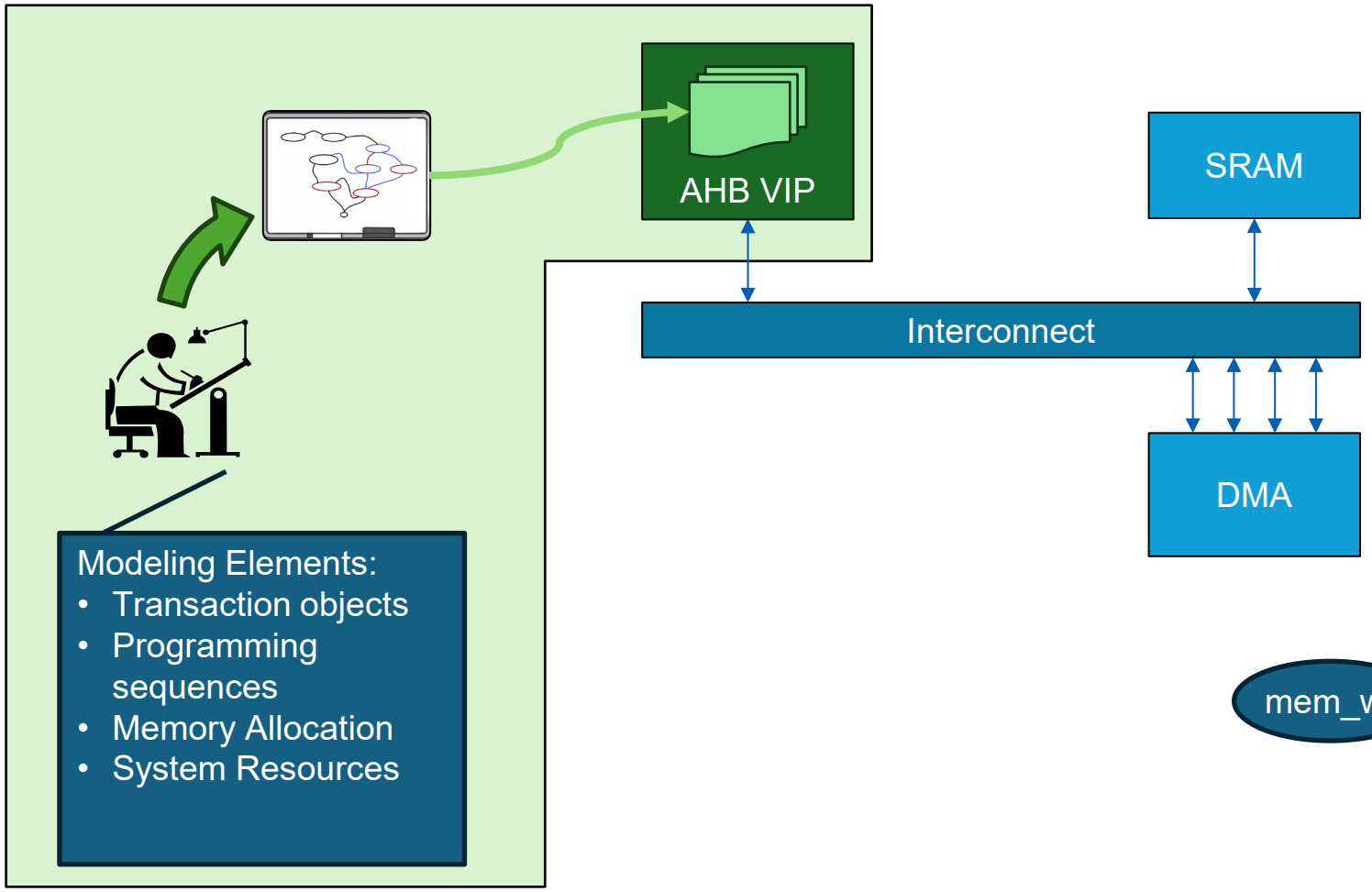
PSS at the Block Level



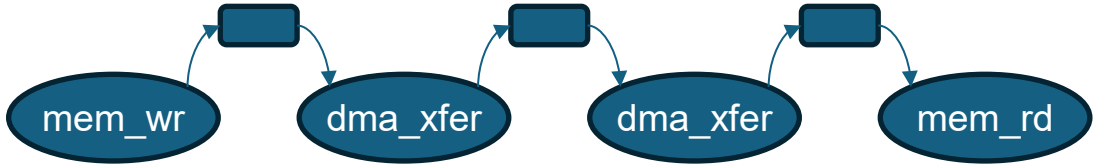
- Critical Behaviors
  - Init, configure
  - tx, rx
- AHB VIP ⊨ the CPU
- UART VIP ⊨ “the world”



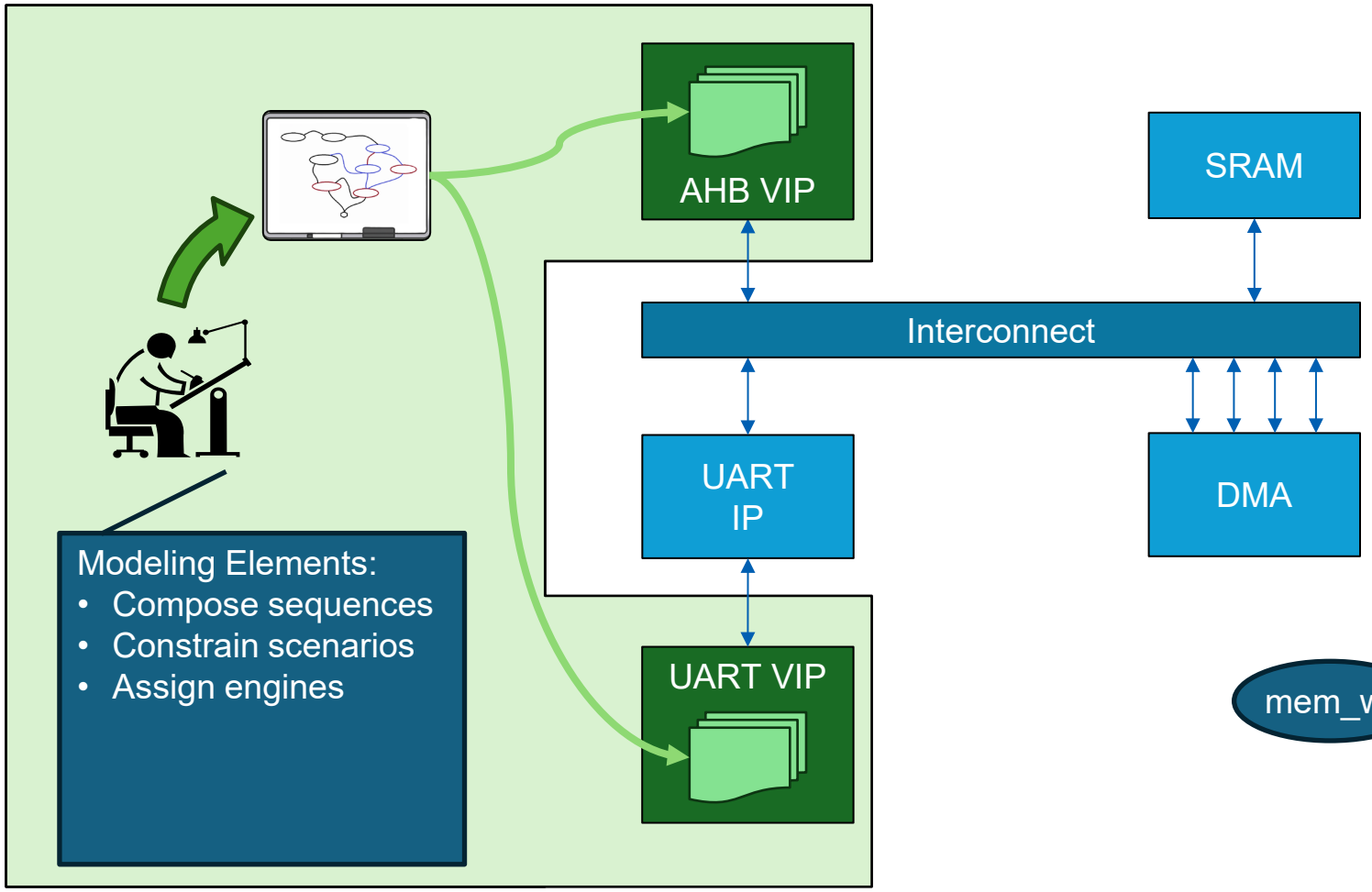
PSS at the Block Level



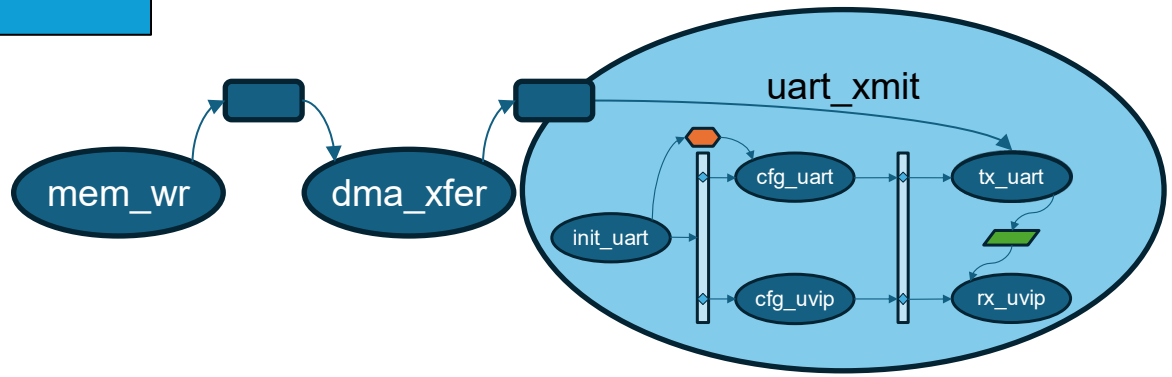
- Critical Behaviors
  - Load/dump RAM
  - Memory copy
  - DMA xfer
- AHB VIP programs everything



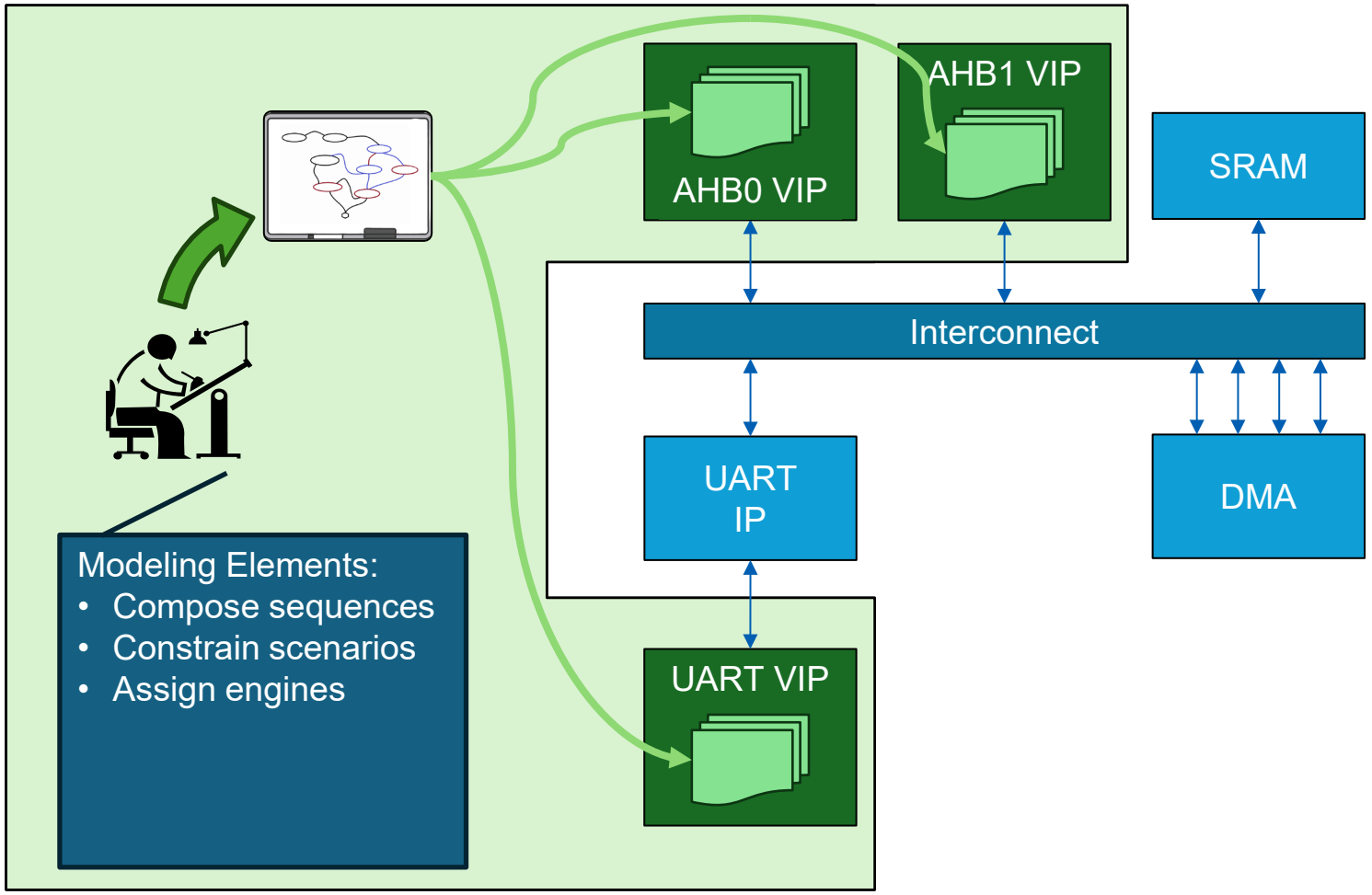
PSS at the (Sub)System Level



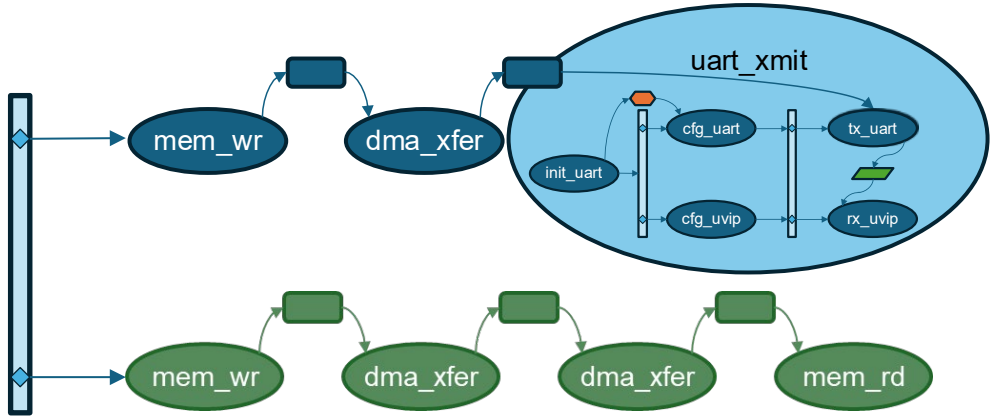
- Critical Behaviors
  - UART-2-SRAM
  - SRAM-2-UART



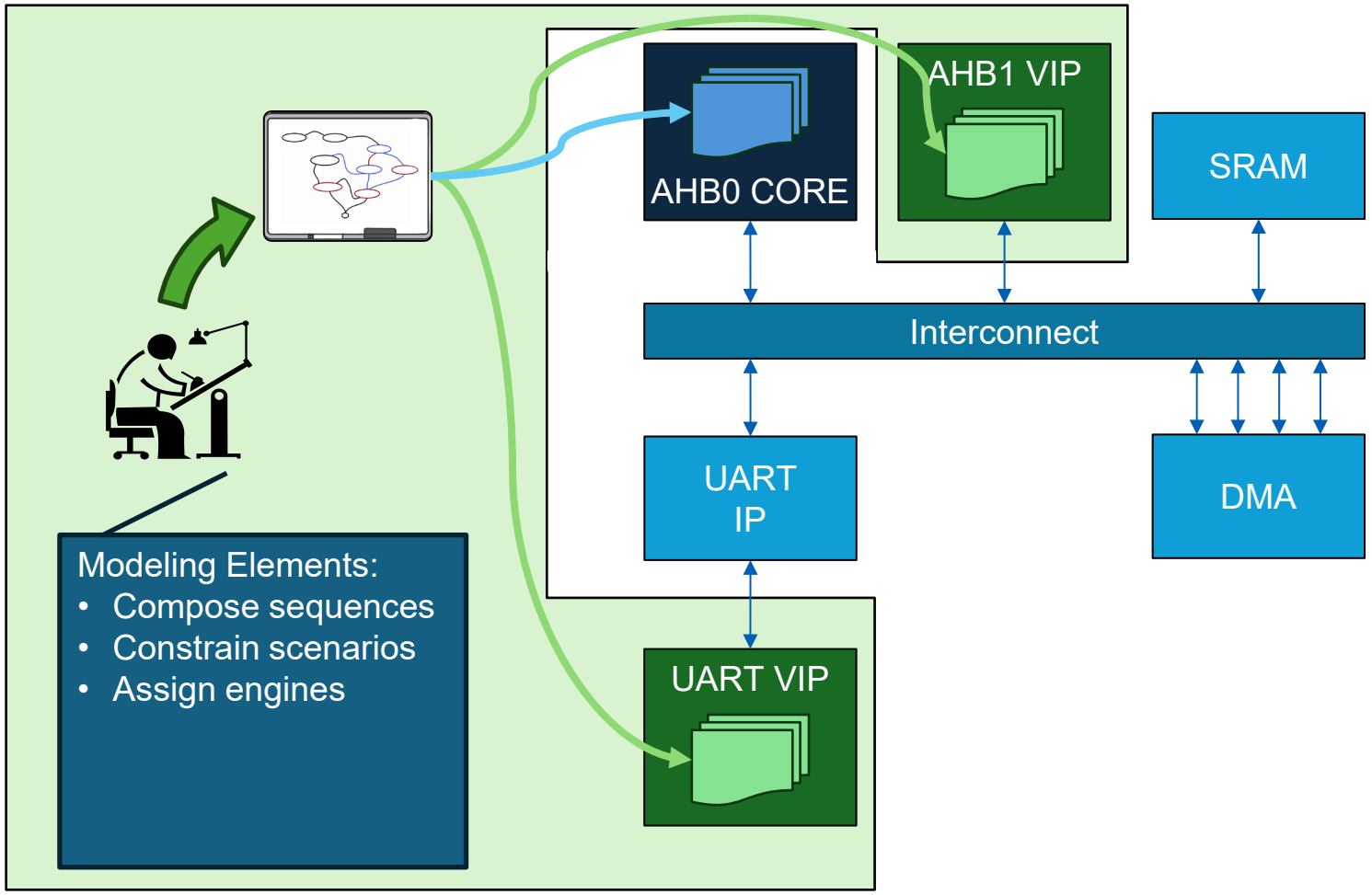
PSS at the (Sub)System Level



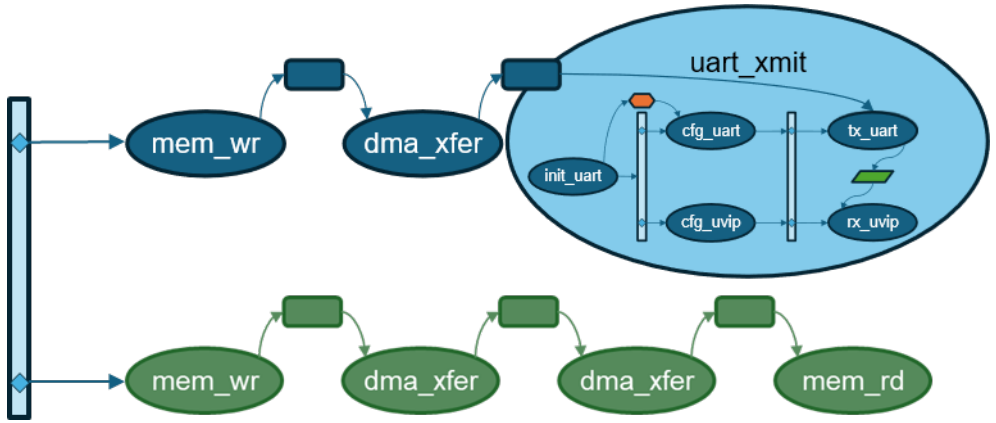
- Critical Behaviors
  - UART-2-SRAM
  - SRAM-2-UART
  - With traffic



PSS at the (Sub)System Level



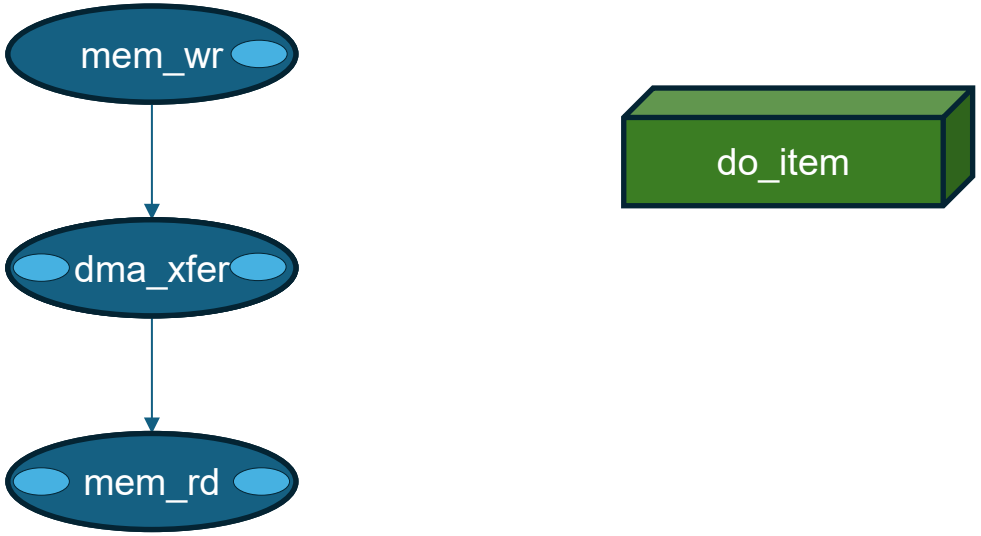
- Critical Behaviors
  - UART-2-SRAM
  - SRAM-2-UART
  - With traffic



PSS vs UVM

UVM relies on transactions

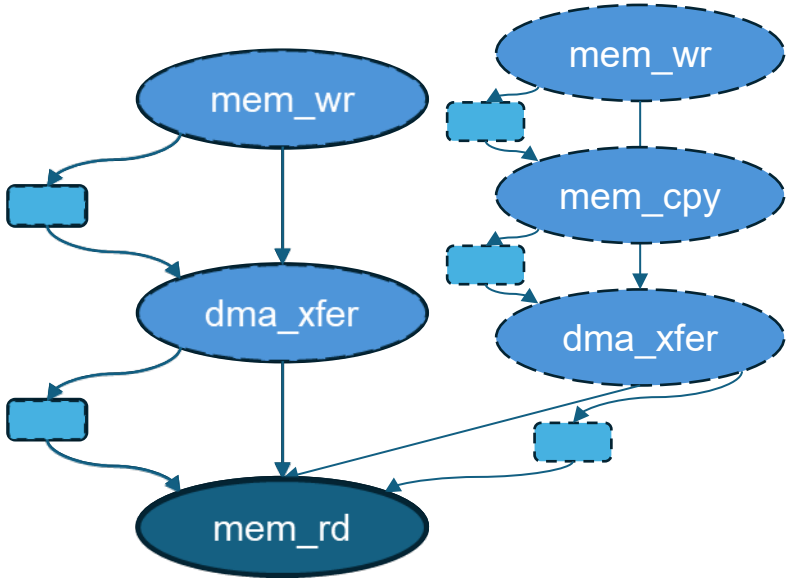
Transactions must be acted upon



- User must manage data flow and constraints

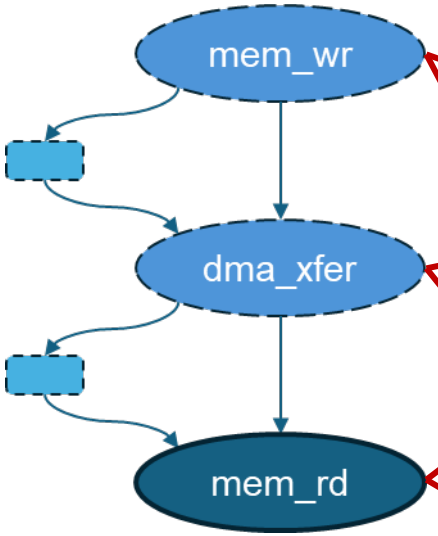
PSS actions encapsulate behaviors

Actions fully define execution semantics



- Tool automatically manages data flow based on action rules

Realization: The Rubber Meets the Road



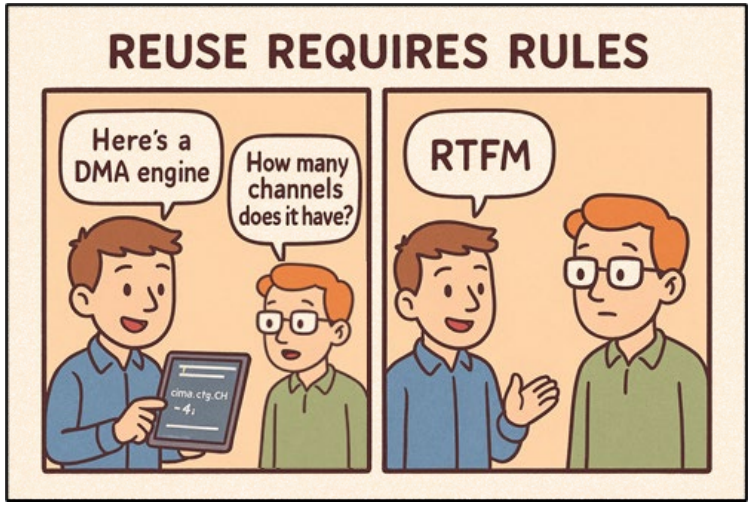

```
class mem_dma_seq extends
   uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(mem_dma_seq)

  virtual task body();
    do_mem_wr(src,data,size);
    do_dma_xfer(src,dst,size,1);
    do_mem_rd(dst,rdata,size);
  endtask
endclass
```

```
int main() {
  mem_wr(srcbuf, data, size);
  dma_xfer(srcbuf, dstbuf, size, 1);
  mem_rd(dstbuf, rdata, size);
  return 0;
}
```

2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Modeling and Automating Device Configuration

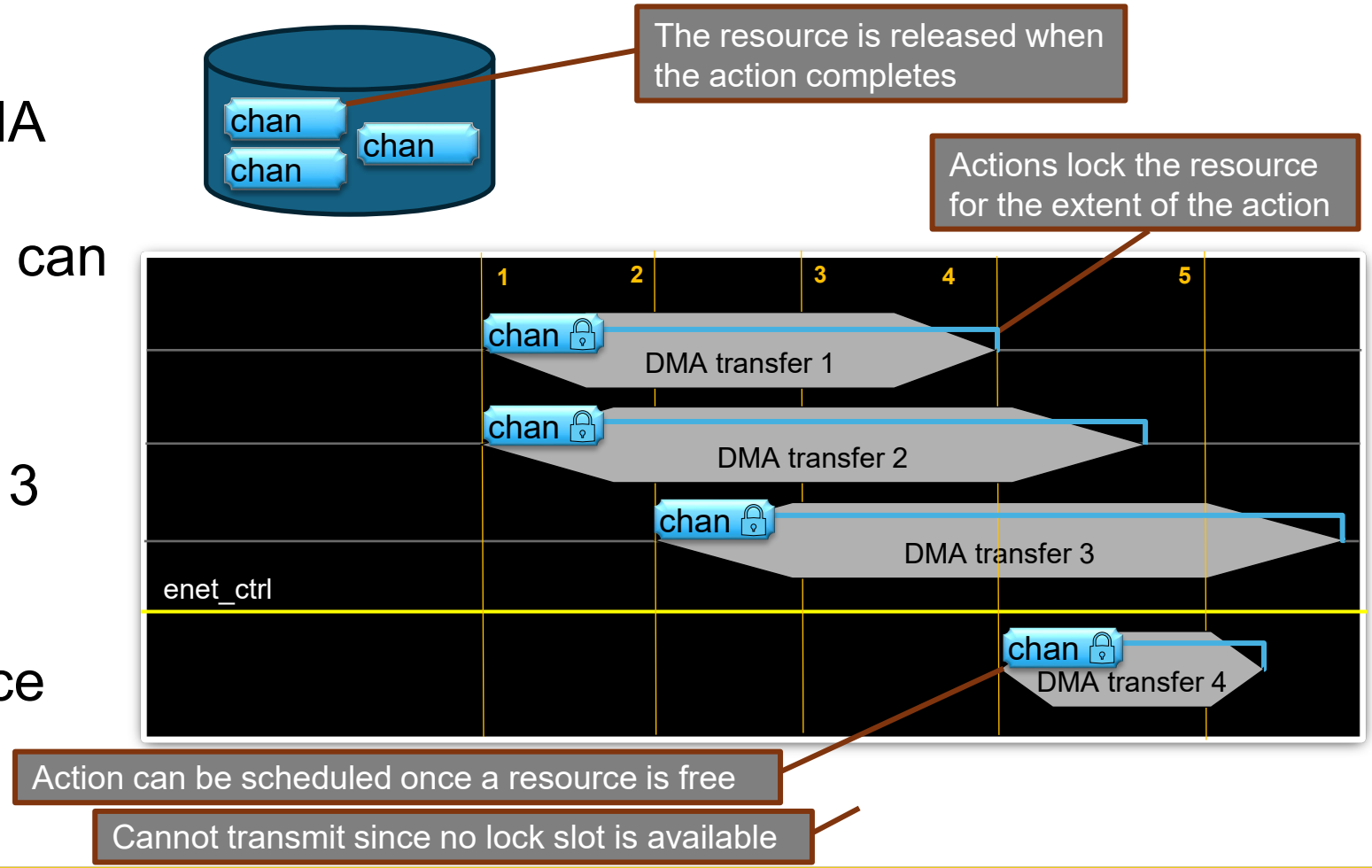
Modeling System Resources



- *Resource* objects in PSS represent a specific piece of the system
- Actions may claim a resource to accomplish a specified behavior
- Resource objects are also grouped into pools
  - Pools define the number of objects available
- Resource pools can be accessed exclusively (***lock***) or non-exclusively (***share***)
- PSS builds system resource information into the model, so you can't use it incorrectly

Scheduling Activity Example: Limited Resources

- Goal:
  - Model a behavior of 3 DMA transfer channels:
  - At most 3 transfer actions can occur simultaneously
- Implementation:
  - Define a resource pool of 3 resource objects
  - Whenever transmit action occurs it locks the resource pool



Device Resource Requirements

- Init – constant allocations
  - One-time device/environment configuration as part of test bring-up
  - Examples: IO pins, pre-programmed test device
- Setup – persistent allocations [init / setup]
  - Device/environment programming that takes long and/or can be reused for multiple activations
  - Examples: video pipes, reconfigurable test device, interrupt-request lines
- Traffic – ad-hoc allocations [setup / traffic / release]
  - Immediate device/environment programming for a single activation
  - Examples: DMA channels, CPU cores, USB endpoints

Making the Right Modeling Decisions

- Is this resource allocation constant? Persistent? Ad-hoc?
- *It depends...*
  - No absolute criterion for a given resource
  - Depends on the scenarios you need to exercise
  - Resources are often constant for one purpose and persistent for another
  - Resources are often ad-hoc for one purpose and persistent for another
- *Be agile with your model!*
  - Start by assuming the simple case
  - Be prepared for simple->complex refinement as information flows in
  - Don't forget complex->simple refactoring when you realize you over-modeled!

A Resource Story

- Consider the following example:
  - CPU interrupt request lines (IRQ ids) serve multiple peripheral devices
  - Can be dynamically assigned to a device using an interrupt controller
- Our exercise –
  - Is this a case of ad-hoc allocation? Persistent? Constant?
  - How is each modeled?
  - How are scenarios specified?
- The modeling patterns are completely general!
  - The same approach can be applied to your favorite restrictive resource
  - The example will use a small number of resources, but the approach applies to any number
  - Even with simplifying assumptions, this can serve as a starting point for many real-life situations

Ad-hoc Allocation – Model

Traffic operation of device A and B need exclusive use of IRQs – use a *lock* !

```
component IOdevA_c {
  action devA_traffic {
    lock irq_r irq_claim;
    rand int in [0..NUM_OF_IRQS-1] irq_id;
    constraint irq_id == irq_claim.instance_id;
  }
}
```

Traffic action claims one **exclusive** resource instance from pool

Can constrain instance index in pool

Instantiation of resource pool, with number of instances

Binding sub-component resource pool references

```
static const int NUM_OF_IRQS = 3;
resource irq_r {};

component pss_top {
  pool [NUM_OF_IRQS] irq_r irq_pool;
  bind irq_pool *;
  //one instance of each device type
  IOdevA_c IOdevA;
  IOdevB_c IOdevB;
}
```

Ad-hoc Allocation – Scenarios

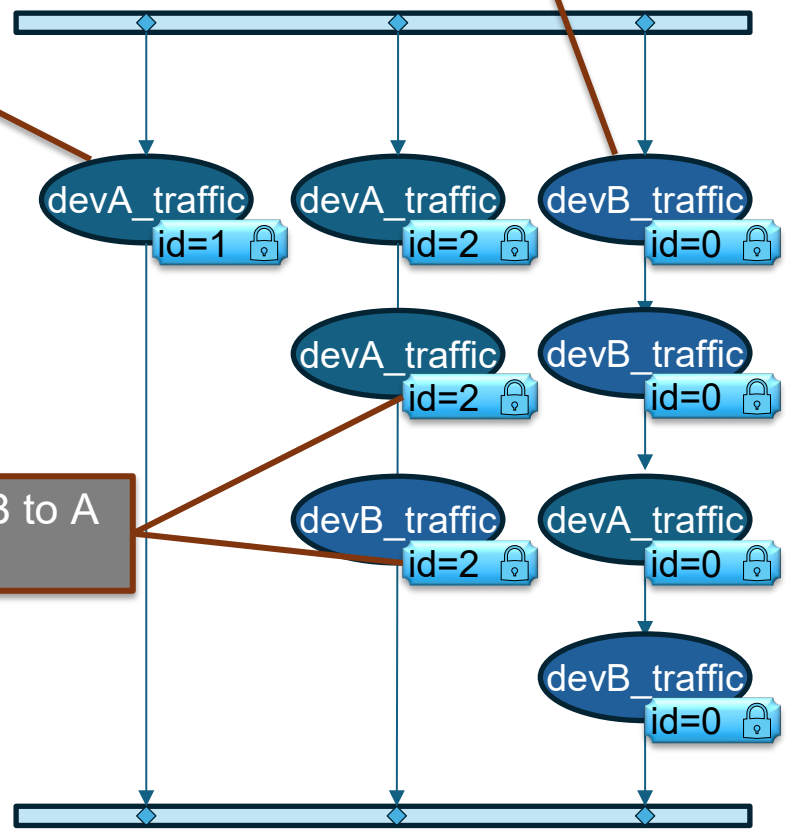
```
action concurrent_traffic {
  activity {
    schedule {
      replicate (8) {
        select {
          do IOdevA_c::devA_traffic;
          do IOdevB_c::devB_traffic;
        }
      }
    }
  }
}
```

Resource instances are *exclusively* allocated for the duration of the actions' executions

Scheduling all actions with same resource instance id *sequentially*

Both devices lock the same resource type

Same irq instance reassigned from B to A
Is that what we need?



Constant Allocation – Model

- Target Requirement: IRQ allocation only once per device
- Put the configuration attribute on the resource object
  - All actions assigned the same resource instance (`instance_id`) "see" the same resource object
  - Must agree on all attribute values

```
static const int NUM_OF_IRQS = 3;
enum irq_assignment_e {none, devA, devB};

resource irq_r {
  // configuration attribute
  rand irq_assignment_e device;
  constraint slot_num == instance_id;
}
component pss_top {
  pool [NUM_OF_IRQS] irq_r irq_pool;
  bind irq_pool *;
  IOdevA_c IOdevA;
  IOdevB_c IOdevB;
}
```

Possible assignment of each resource instance

```
component IOdevA_c {
  action devA_traffic {
    lock irq_r irq_claim;
    rand int in [0..NUM_OF_IRQS-1] irq_id;
    constraint irq_id == irq_claim.slot_num;
    constraint irq_claim.device == devA;
  }
}
```

```
component IOdevB_c {
  action devB_traffic {
    lock irq_r irq_claim;
    rand int in [0..NUM_OF_IRQS-1] irq_id;
    constraint irq_id == irq_claim.slot_num;
    constraint irq_claim.device == devB;
  }
}
```

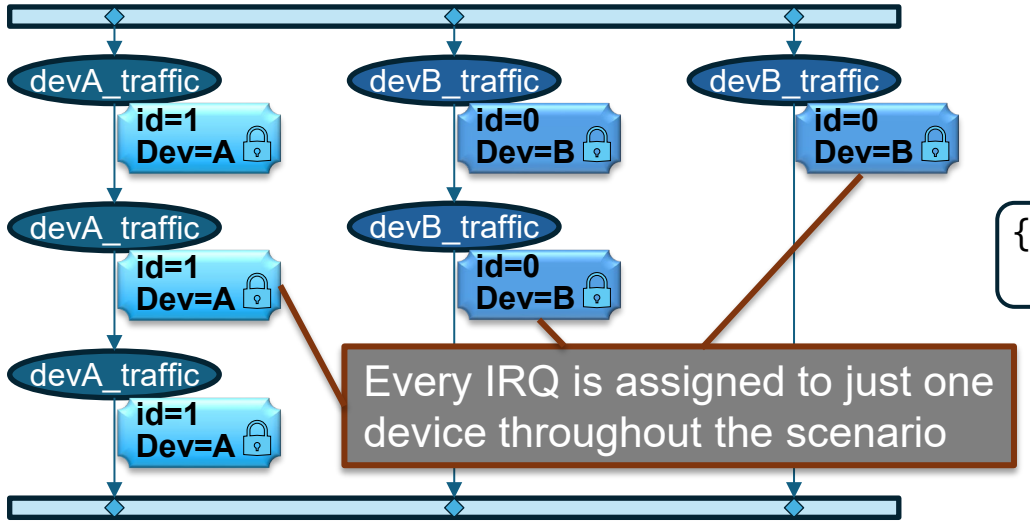
Cannot assign the same resource to actions of different device

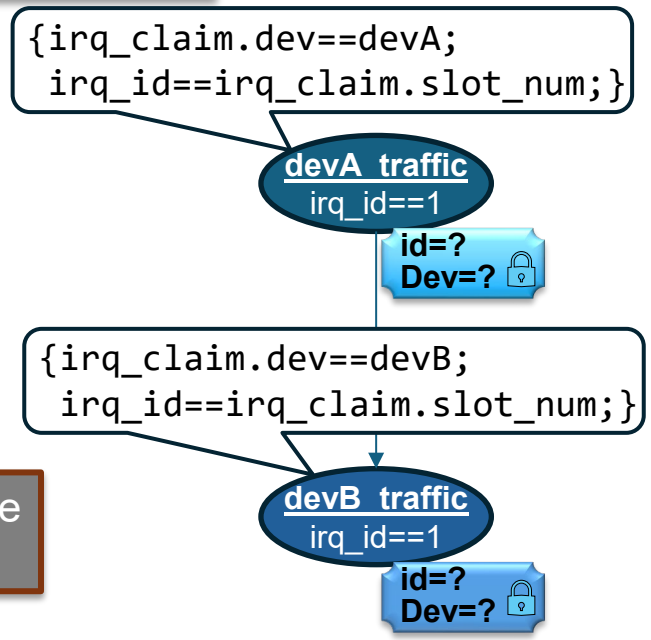
Constant Allocation - Scenarios

```
action concurrent_traffic {
  activity {
    schedule {
      replicate (6) {
        select {
          do IOdevA_c::devA_traffic;
          do IOdevB_c::devB_traffic;
        }
      }
    }
  }
}
```

```
action illegal_scenario {
  activity {
    do IOdevA_c::devA_traffic with {irq_id == 1}
    do IOdevB_c::devB_traffic with {irq_id == 1}
  }
}
```

Is that what we need?
What if IRQ assignment can be re-programmed during a test ?





State Variables: Persistent Device Configuration

Device A can be configured in some mode for multiple traffic activations, and subsequently reconfigured

- Use a *state variable*!

```
enum devA_speed_e {SLOW, FAST};
enum devA_format_e {X, Y, Z};

state devA_config_s {
  rand devA_speed_e speed;
  rand devA_format_e format;
  constraint initial -> speed == SLOW;
  constraint initial -> format == X;
}
```

State object type representing aggregate configuration state

Specific configuration attributes

Initial value constraints


```
component IOdevA_c {
  ...
  pool devA_config_s devA_config_var;
  bind devA_config_var *;
}
```

State variable pool and binding

State Variables: Persistent Device Configuration

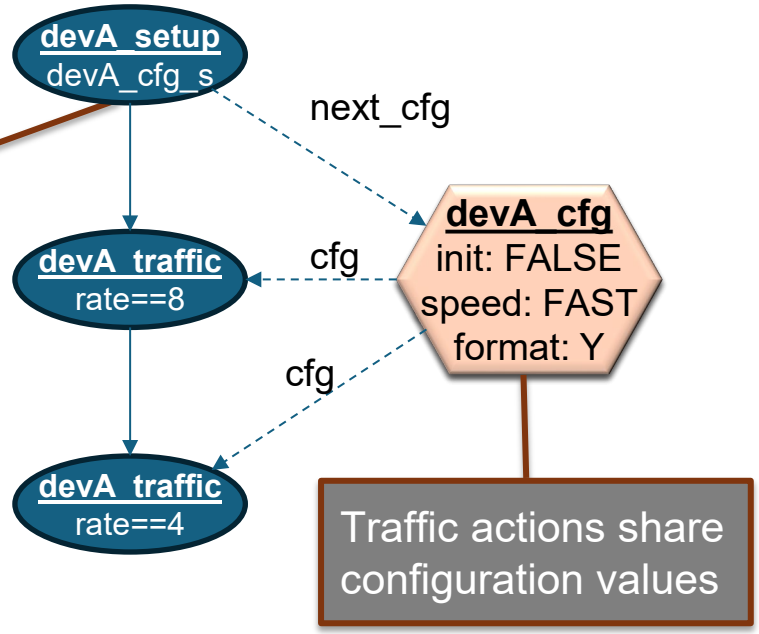
```
component IOdevA_c {
  pool devA_config_s devA_config_var;
  bind devA_config_var devA_setup.next_cfg;
  bind devA_config_var devA_traffic.cfg;

  action devA_setup {
    output devA_config_s next_cfg;
  }

  action devA_traffic {
    rand int in [1..2,4,8] rate;
    input devA_config_s cfg;
    constraint (rate == 8) -> cfg.speed == FAST;
  }
}
```

Setup action establishes a new configuration

One setup per any number of traffic actions

Traffic action "reads" (depends on) configuration state

Constraints on configuration attributes



Persistent Allocation – Structure

Back to our IRQ story... Now you realize that IRQ assignment can be re-programmed and should apply to *multiple* activations – use a *state variable*!

- Possible assignment of each resource instance
- Number of resource instances
- State object type representing aggregate state
- One state-value attribute per each resource
- Initial value for each resource instance
- State variable pool and binding

```
enum irq_assignment_e {none, devA, devB};
static const int NUM_OF_IRQS = 3;
state irq_config_s {
  rand irq_assignment_e irq_map[NUM_OF_IRQS];
  constraint foreach (it: irq_map) {
    initial -> it == none;
  }
}
```

irq_cfg
init: TRUE
map[3]: ∅,∅,∅

```
extend component pss_top {
  pool irq_config_s irq_config_var;
  bind irq_config_var *;
  ...
}
```

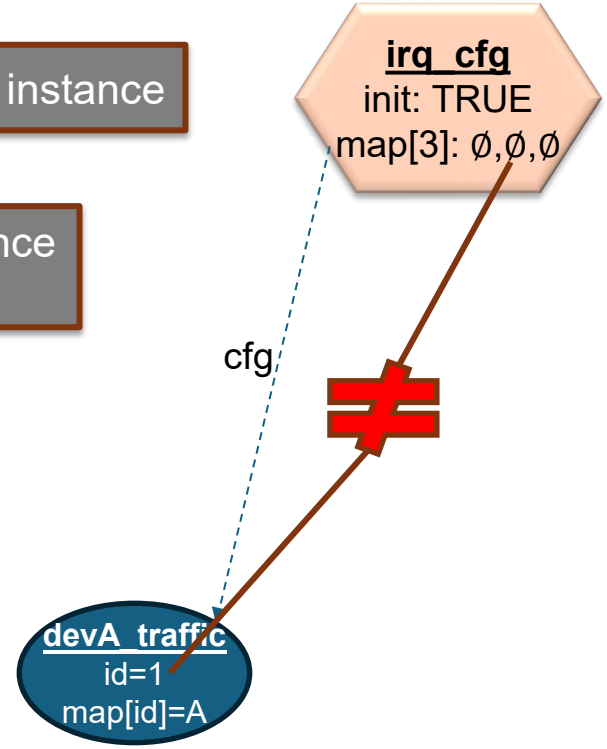
Persistent Allocation – Traffic Dependency

```
action devA_traffic {
  input irq_config_s irq_cfg;


  rand int in [0..NUM_OF_IRQS-1] irq_id;
  constraint foreach (it: irq_cfg.irq_map[i]) {
    i == irq_id -> irq_cfg.irq_map[i] == devA;
   }
}
```

Traffic action "reads" resource-assignment state

Traffic action "chooses" a resource instance

Choice must be of a resource instance that is assigned to this device



Persistent Allocation – Setup

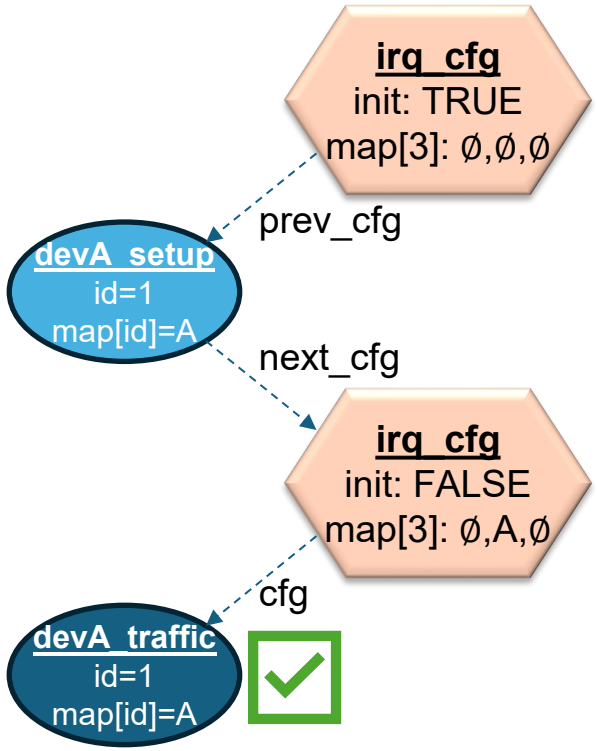
```
action devA_setup {

  input irq_config_s prev_cfg;
  output irq_config_s next_cfg;

  rand int in [0..NUM_OF_IRQS-1] irq_id;

  constraint foreach (it: prev_cfg.irq_map[i]) {
        i == irq_id ?
             next_cfg.irq_map[i] == devA:
             next_cfg.irq_map[i] == it;
  }
}
```

Setup action "reads" previous resource-assignment state and "writes back" the new state

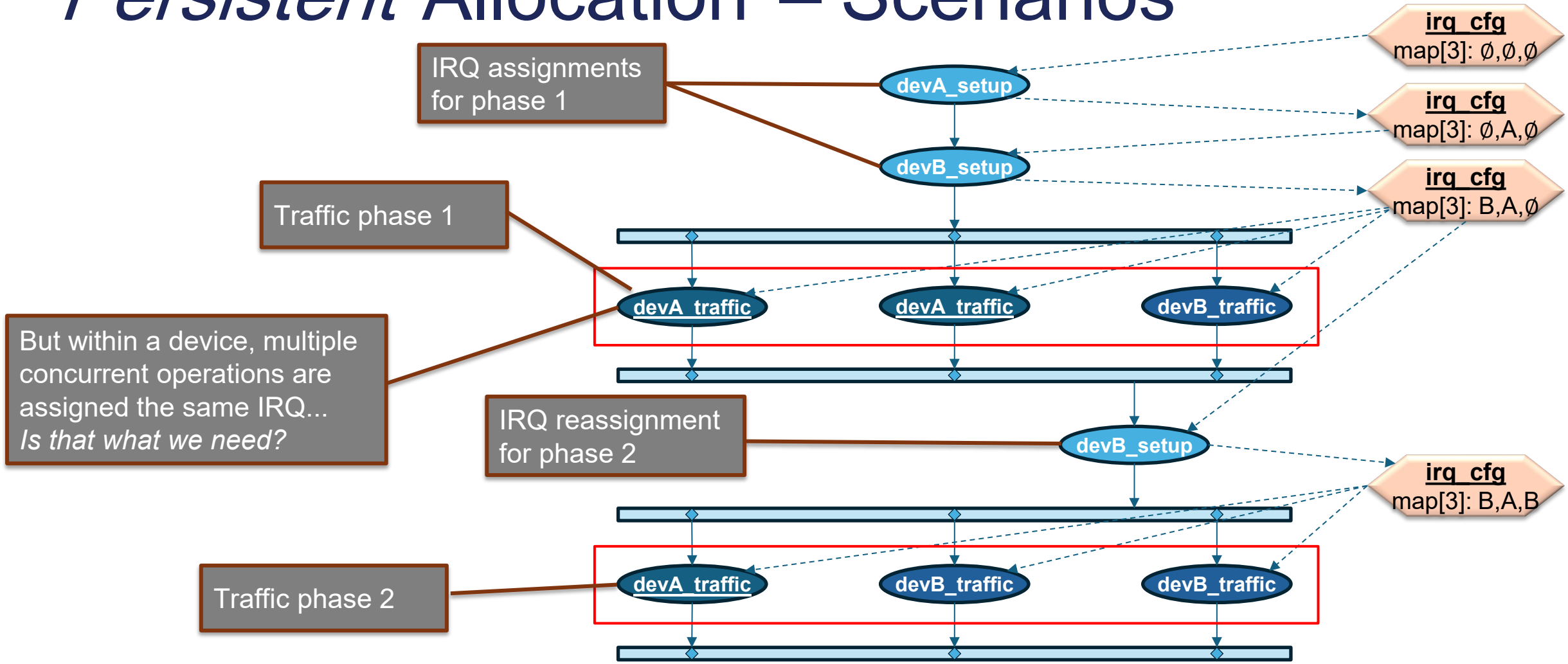
Setup action "chooses" a resource instance

The chosen instance changes value to be assigned to this device

Assignment of all other resources remains unchanged!



Persistent Allocation – Scenarios

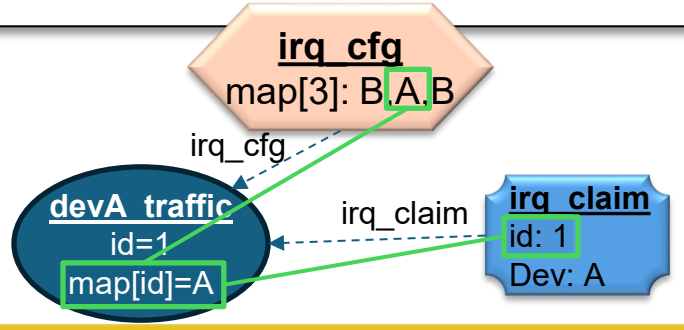


Persistent Exclusive Allocation – Model

IRQ must be uniquely assigned to a device for multiple activations, and exclusively to each traffic operation – use both lock and state variable!

```
extend component pss_top {
  pool [NUM_OF_IRQS] irq_r irq_pool;
  pool irq_config_s irq_config_var;

  bind irq_config_var *;
  bind irq_pool *;

  IOdevA_c IOdevA;
  IOdevB_c IOdevB;
};
```

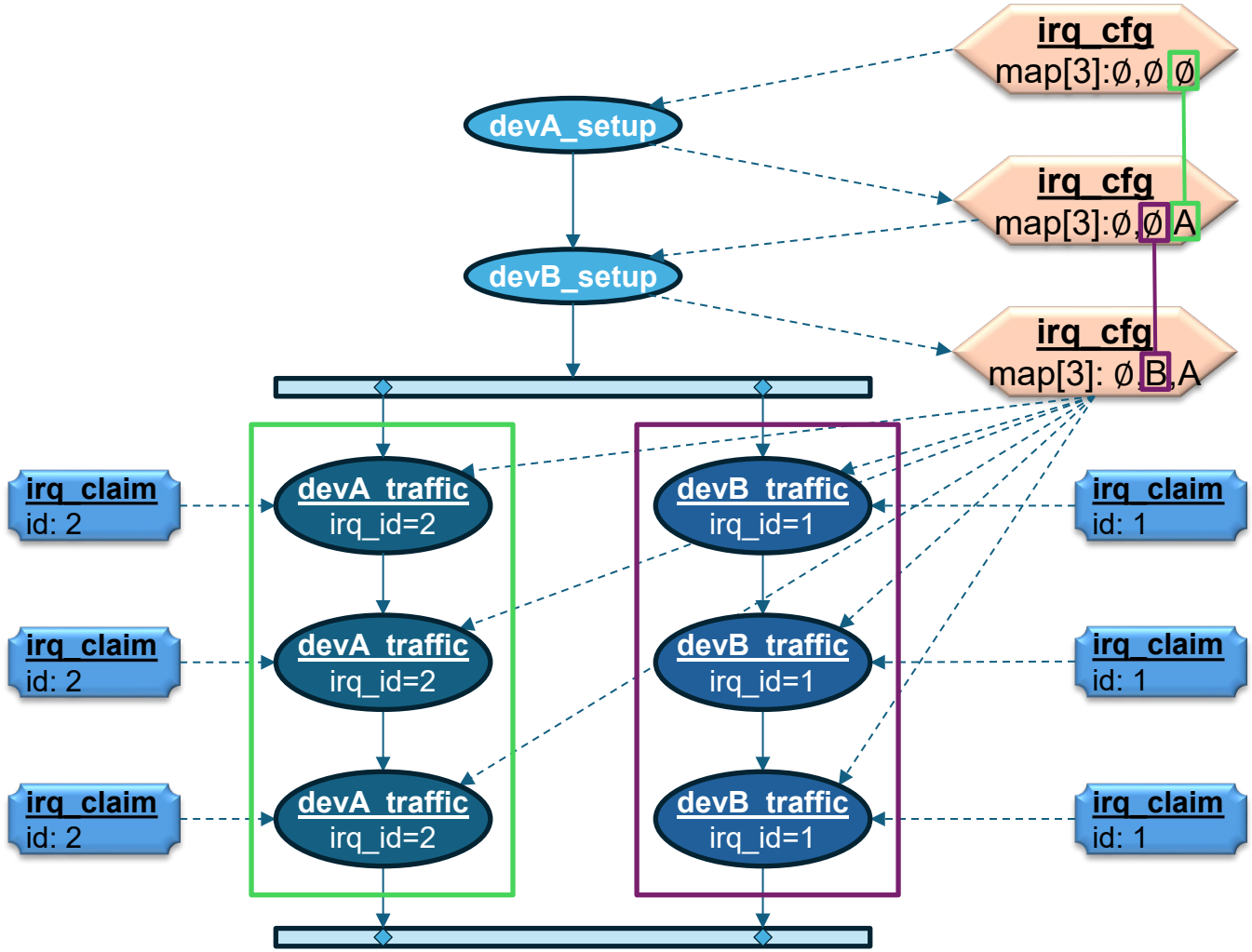
```
action devA_traffic {
  input irq_config_s irq_cfg;
  lock irq_r irq_claim;
  rand int in [0..NUM_OF_IRQS-1] irq_id;
  constraint irq_id == irq_claim.instance_id;
  constraint foreach (it: irq_cfg.irq_map[i]) {
    i == irq_id -> irq_cfg.irq_map[i] == devA;
  }
};
```



Persistent Exclusive Allocation – Scenarios

```
action concurrent_traffic {
  rand int num_traffic;
  constraint default num_traffic == 6;
  activity {
    schedule {
      replicate (num_traffic) {
        select {
          do IOdevA_c::devA_traffic;
          do IOdevB_c::devB_traffic;
        }
      }
    }
  }
}
```

```
action multi_phased_traffic {
  activity {
    sequence {
      do IOdevA_c::devA_setup;
      do IOdevB_c::devB_setup;
      do concurrent_traffic;
    }
  }
}
```



Constant Allocation Alternative Approach with States

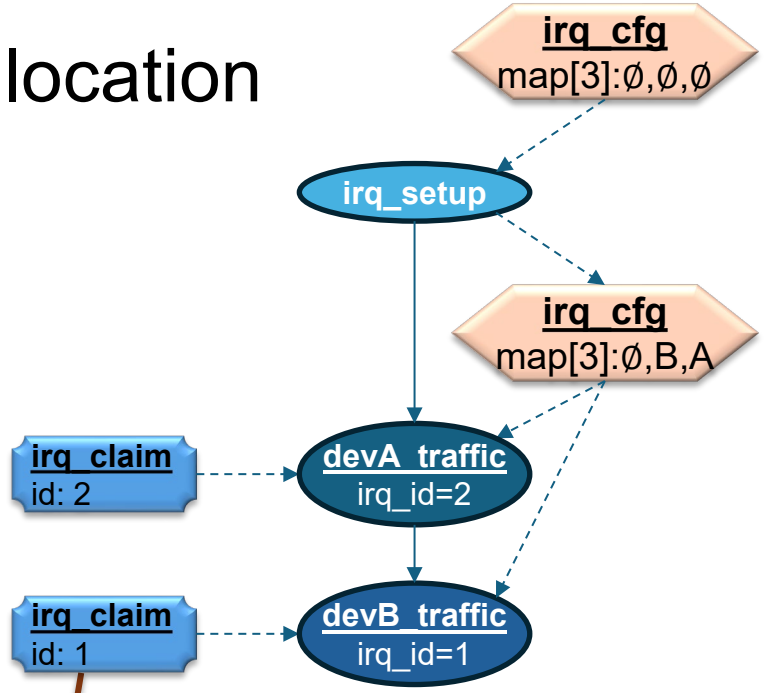
Finally, you realize that in the target platform IRQ allocation must be done only once per test

- Use initial constraint!

```
extend pss_top {
  pool irq_config_s irq_config_var;
  bind irq_config_var *;
  action irq_setup {
    input irq_cfg_s initial_irq_cfg;
    output irq_cfg_s irq_cfg;
    constraint initial_irq_cfg.initial == true;
    ...
  }
}
```

Single setup action that assigns all possible IRQ instances

Input must be in **initial** state, therefore, setup can take place only once!



Cannot use same IRQ instance for two different devices in the same test!

2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Realizing PSS scenarios in UVM and C

PSS Action Realization: Target Templates

String literal is not recognized nor syntax-highlighted by a simple editor

Model values passed in via **{{moustache}}** notation

```
extend action dma_c::dma_xfer {
  exec body C = """
    dma_program({{channel.instance_id}},{{src.size}},
                {{src.addr}},{{dst.addr}},INCR);
    dma_activate({{channel.instance_id}});
    while (!dma_xfer_done({{channel.instance_id}}))
      yield();
  """;
}
```

Need a separate exec block for each target implementation

```
extend action dma_c::dma_xfer {
  exec body SV = """
    dma_program({{channel.instance_id}},{{src.size}},
                {{src.addr}},{{dst.addr}},INCR);
    dma_activate({{channel.instance_id}});
    while (!dma_xfer_done({{channel.instance_id}}))
      #0;
  """;
}
```

PSS Action Realization: Foreign Procedures

Method implementation determined by which package is imported

```
extend action dma_c::dma_xfer {
  exec body {
    do_dma(channel.instance_id,src.size,
           src.addr,dst.addr,INCR);
  }
}
```

```
package dma_sv_func_pkg {
  function void do_dma(int channel, int size,
                       int src_addr, int dst_addr, int mode);
  import target SV function do_dma;
}
```

```
task automatic do_dma(int channel, int size,
                      int src_addr, int dst_addr);
  dma_program(channel, size, src_addr, dst_addr);
  dma_activate(channel);
  while (!dma_xfer_done(channel))
    #0;
endtask
```

```
package dma_c_func_pkg {
  function void do_dma(int channel, int size,
                       int src_addr, int dst_addr, int mode);
  import target C function do_dma;
}
```

Still need to implement the algorithm multiple times

```
void do_dma(int channel, int size,
            int src_addr, int dst_addr) {
  dma_program(channel,size,src_addr,dst_addr);
  dma_activate(channel);
  while (!dma_xfer_done(channel))
    yield();
}
```

PSS Action Realization: Procedural Execs

Code is checked for syntax, type correctness, and all static semantic rules

```
extend action dma_c::dma_xfer {
  exec body {
    dma_program(channel.instance_id,src.size,
                src.addr,dst.addr,INCR);
    dma_activate(channel.instance_id);
    while (!dma_xfer_done(channel.instance_id))
      yield;
  }
}
```

```
package dma_sv_funcs_pkg {
  function void dma_program(int channel, int size,
                            int src_addr, int dst_addr);
  import target SV function dma_program;
  function void dma_activate(int channel);
  import target SV function dma_activate;
  function bit dma_xfer_done(int channel);
  import target SV function dma_xfer_done;
}
```

```
package dma_c_funcs_pkg {
  function void dma_program(int channel, int size,
                            int src_addr, int dst_addr);
  import target C function dma_program;
  function void dma_activate(int channel);
  import target C function dma_activate;
  function bit dma_xfer_done(int channel);
  import target C function dma_xfer_done;
}
```

Each procedural (sub)method must still be implemented multiple times

PSS Action Realization: Procedural Execs

```
component dma_c {
  reggrp1 regs;
  transparent_addr_region_s<> regs_region;
  exec init_up {
    regs_region.size = 0x1000;
  }
}
```

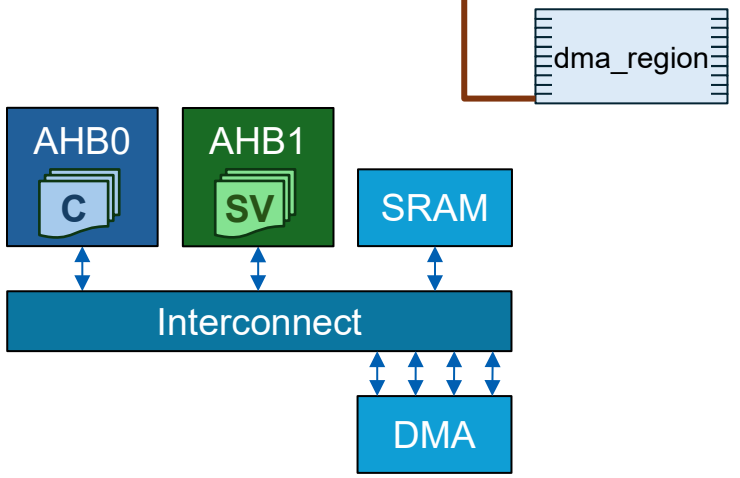
```
extend action dma_c::dma_xfer {
  exec body {
    comp.regs.src_reg[channel.instance_id].write(src.addr);
    comp.regs.dst_reg[channel.instance_id].write(dst.addr);
    comp.regs.size_reg[channel.instance_id].write(src.size);
    comp.regs.ctrl_reg[channel.instance_id].write(INCR);
    comp.regs.go_reg[channel.instance_id].write(1);
    while (!comp.regs.status[channel.instance_id].read())
      yield;
  }
}
```

Full algorithm implemented once in PSS

```
comp.regs.src_reg[channel.instance_id].write(src.addr);
```

Calculate register address offset

Base register handle



PSS Action Realization: Registers

```
component pss_top {
  import addr_reg_pkg::*;
  import my_exec_pkg::*;
  contiguous_addr_space_c<> sys_mem;
  dma_c dma;
  exec init_up {
    addr_handle_t h;
    dma.regs_region.base_addr = 0x80000;
    h = sys_mem.add_nonallocatable_region(dma.regs_region);
    dma.regs.set_handle(h);
  }
```

```
write32(0x80000,0x12340);
write32(0x80004,0x43210);
...
```

```
comp.regs.src_reg[channel.instance_id].write(src.addr);
```

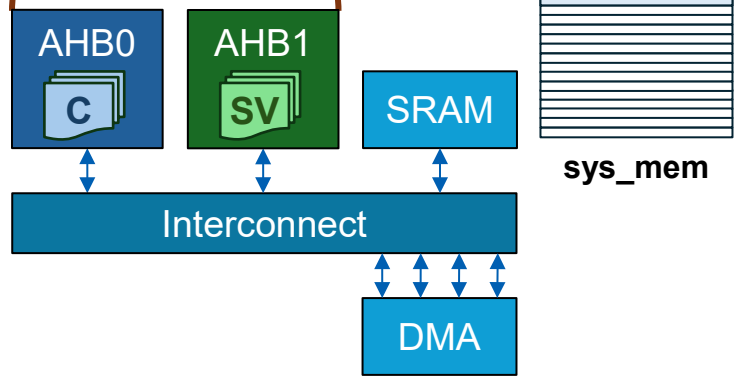
Calculate register address offset

PSS Tool converts reg access to primitive

Base register handle

```
wr h = make handle from handle(h,offset);
write32(wr_h,src.addr);
```

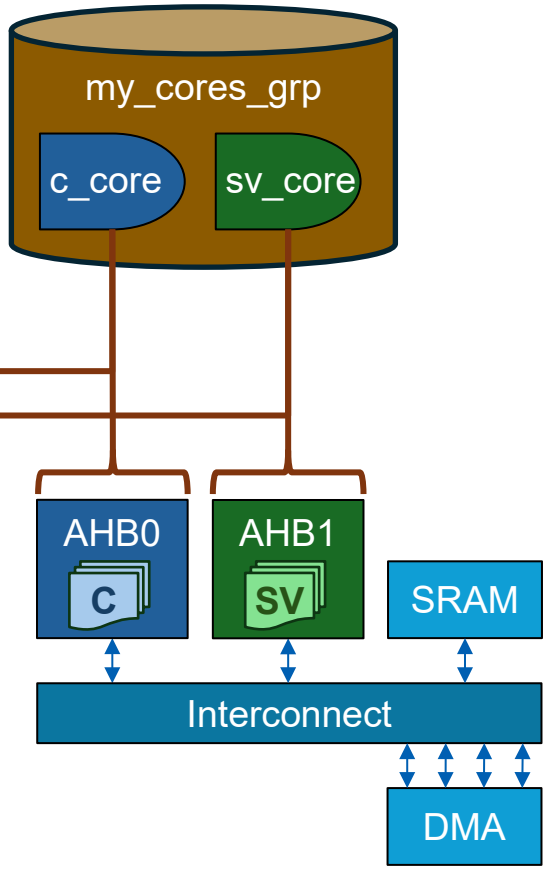
Which target element will execute the write32() call?



PSS Action Realization: Executors

```
extend component pss_top {
  my_cores_group_c core_group;
  extend action entry {
    rand executor_claim_s<core_trait_s> core;
    activity {
      do dma_c::dma_xfer with {core.trait.core_id == 0;};
      do dma_c::dma_xfer with {core.trait.core_id == 1;};
    }
  }
}
```

```
  component my_cores_group_c : executor_group_c<core_trait_s> {
    my_sv_executor_c sv_core;
    my_c_executor_c c_core;
    exec init_down {
      sv_core.trait.core_id = 0;
      add_executor(sv_core);
      c_core.trait.core_id = 1;
      add_executor(c_core);
    }
  }
```



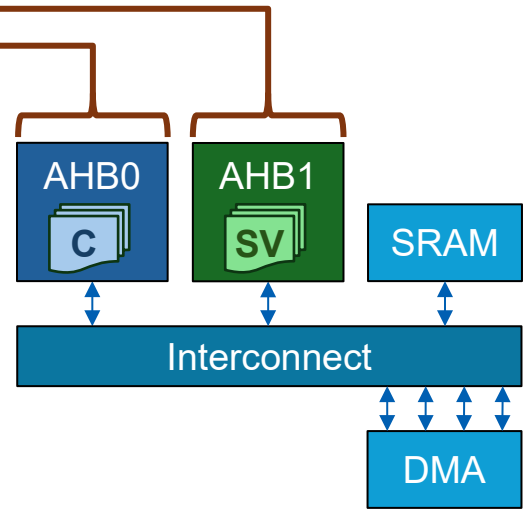
PSS Action Realization: Executors

```
extend component pss_top {
  my_cores_group_c core_group;
  extend action entry {
    rand executor_claim_s<core_trait_s> core;
    activity {
      do dma_c::dma_xfer with {core.trait.core_id == 0;};
      do dma_c::dma_xfer with {core.trait.core_id == 1;};
    }
  }
}
```

```
function void write32(addr_handle_t hndl) {
  if (executor() != null ) {
    executor().write32(hndl, data);
  } else {
      //use default implementation
  }
}
```

PSS Tool effectively implements this transformation

```
component my_sv_executor_c : executor_c<core_trait_s> {
  import target SV function void my_sv_write_word(bit[64] addr, bit[32] data);

  function void write32(addr_handle_t hndl, bit[32] data) {
    my_sv_write_word(addr_value(hndl), data);
  }
}
component my_c_executor_ c : executor_c<core_trait_s> {
  import target C function void my_c_write_word(bit[64] addr, bit[32] data);

  function void write32(addr_handle_t hndl, bit[32] data) {
    my_c_write_word(addr_value(hndl), data);
  }
}
```



PSS Action Reuse: Export

```
export target dma_c::dma_xfer (bit[32] src,dst,size; bit[4] chan);
```

```
extend action dma_c::dma_xfer {
  exec body {
    comp.regs.src_reg[channel.instance_id].write(src.addr);
    comp.regs.dst_reg[channel.instance_id].write(dst.addr);
    comp.regs.size_reg[channel.instance_id].write(src.size);
    comp.regs.ctrl_reg[channel.instance_id].write(INCR);
    comp.regs.go_reg[channel.instance_id].write(1);
    while (!comp.regs.status[channel.instance_id].read())
      yield;
  }
}
```

SV

```
task dma_xfer_task (bit[31:0] src,dst,size,
                    bit[3:0] chan);
  my_sv_write_word('h0020,src);
  my_sv_write_word('h0024,dst);
  my_sv_write_word('h0028,size);
  my_sv_write_word('h002c,chan);
  my_sv_write_word('h0030,`b1);
  while (!my_sv_read_word('h0034))
    #0;
endtask
```

C

```
void dma_xfer_task (int src, int dst,
                    int size, int chan) {
  my_c_write_word(0x0020,src);
  my_c_write_word(0x0024,dst);
  my_c_write_word(0x0028,size);
  my_c_write_word(0x002c,chan);
  my_c_write_word(0x0030, 1);
  while (!my_c_read_word(0x0034))
    yield();
}
```

2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

PSS is the Answer

Why PSS?

- PSS lets you model verification intent once, independent of the target platform
- PSS is layered and flexible to let you tune your intent as requirements grow (and shrink)
- PSS provides several ways to map the model to implementation
  - Which you use depends on your needs
  - Executors let you map action traversals to specific testbench elements
- PSS simplifies scheduling and data flow between actions

2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Q & A

2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Thank You!