



Portable Stimulus Standard Tutorial

Hiroshi Hosokawa

Canon U.S.A.



Agenda

- **PSS Introduction**
- **Developing Reusable Test Content at Block Level**
- **Sub-system and SoC-level testing**
- **PSS new features and conclusion**



PSS Introduction

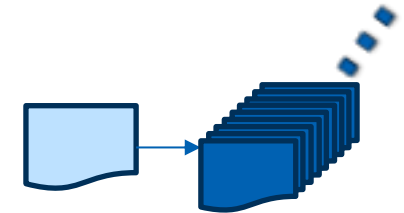
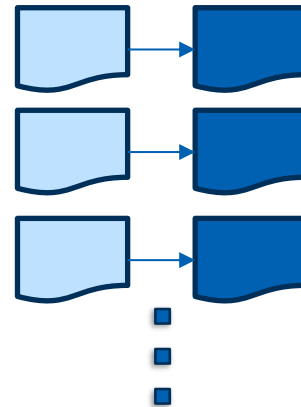
Methodology Shifts Require New Thinking



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

One piece of code
per test

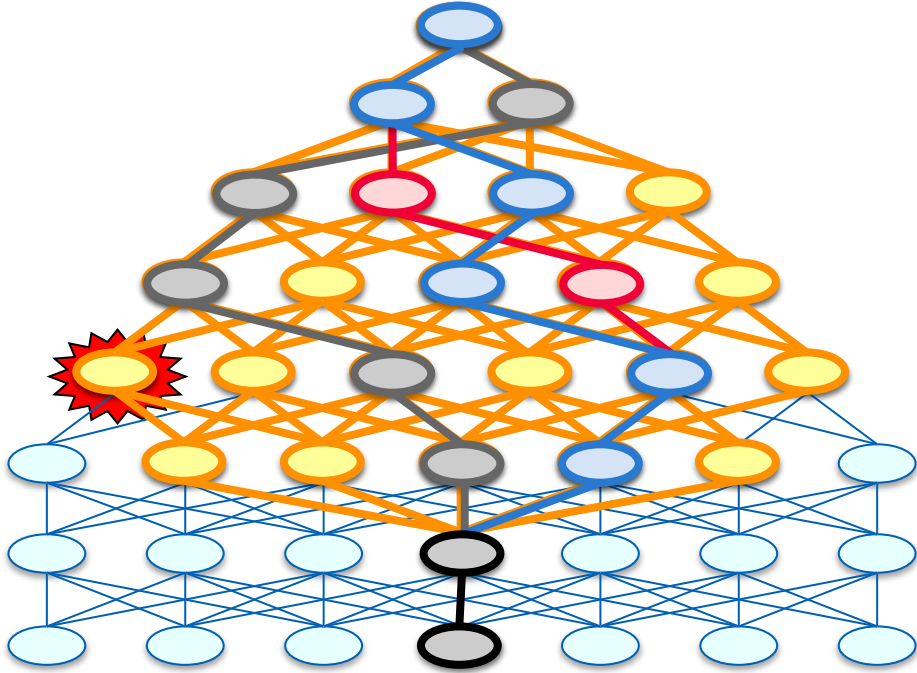
Multiple tests per
piece of code



- **Constrained-Random requires Functional Coverage to know what happened**

PSS is Declarative

Brings Constrained-Random Generation to the Scenario Level

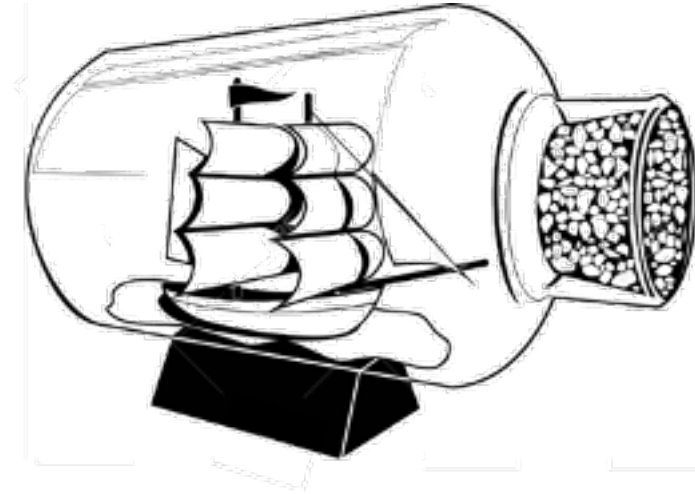


- **Higher level of abstraction**
 - Concise models
 - Describe a much larger set of tests
- **Specifies rules to define the set of possible scenarios**
 - Scheduling constraints between actions
 - Data flow requirements between actions
 - Data constraints
 - Target-specific resource constraints
- **Tool generates code to execute on Target Platform**
 - Each unique solution is effectively a directed test
 - May infer action executions to meet rule requirements
 - “Overlaying” tests effectively covers the desired test space

What is a Portable Stimulus Model?

The
Abstract
Model

- *What* does it do



The
Realization
Layer

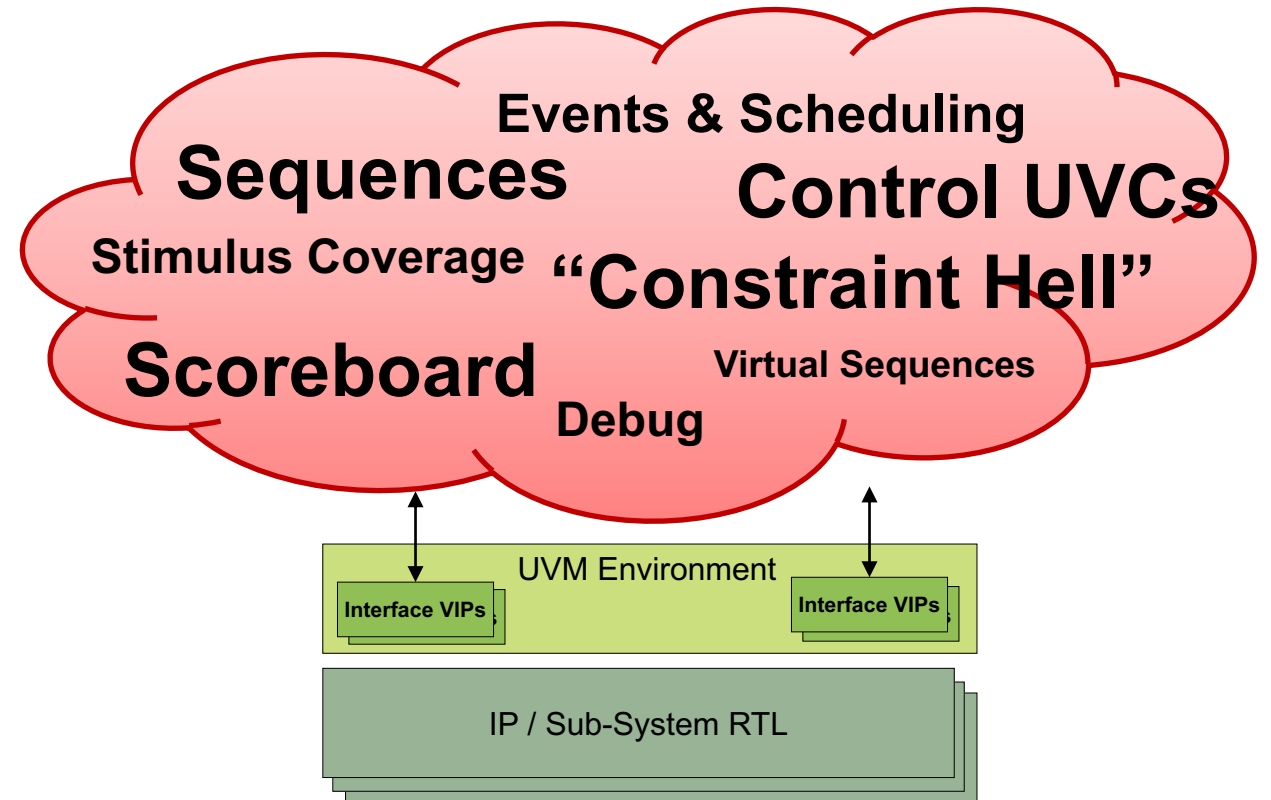
- *How* does it do what it does



Concise Language to Specify Verification Intent

A complement to UVM, not a replacement

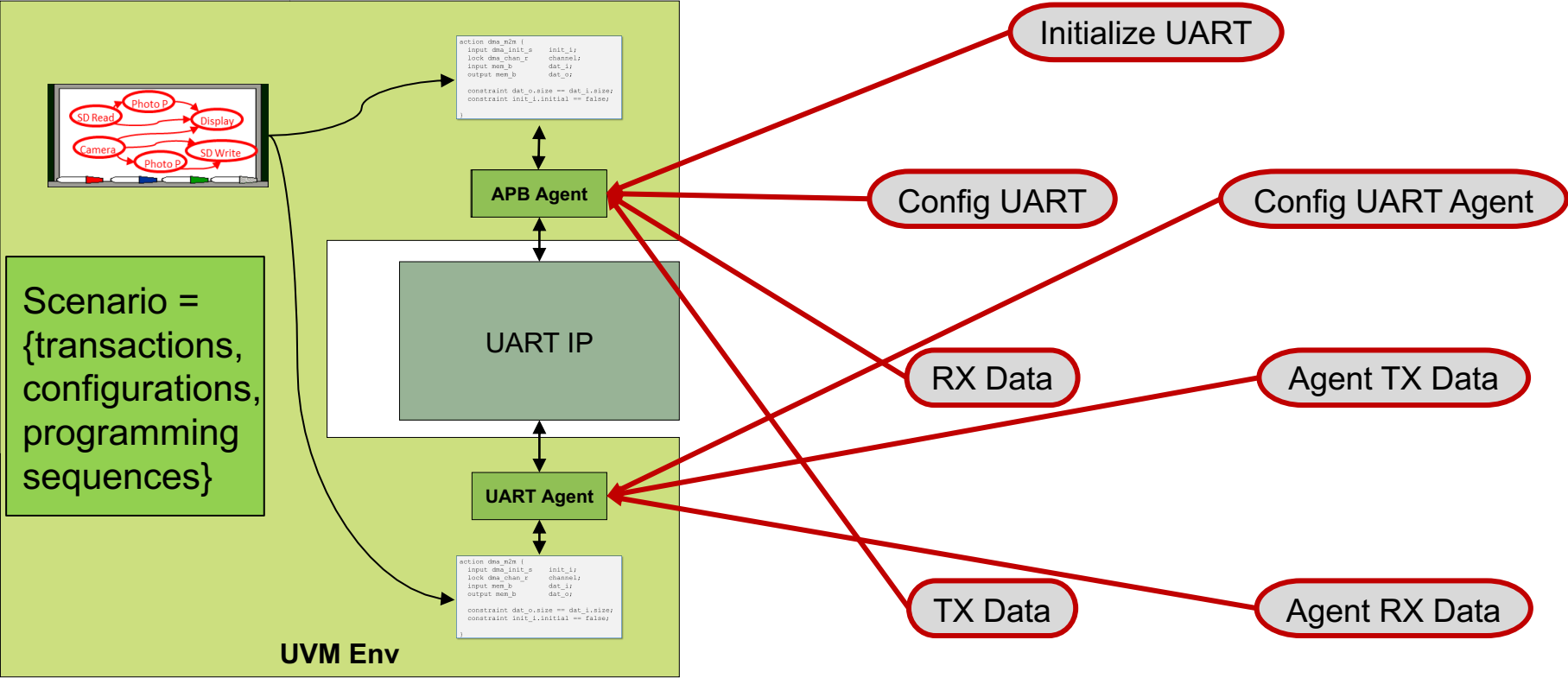
PSS is a stimulus language



Concise Language to Specify Verification Intent

A complement to UVM, not a replacement

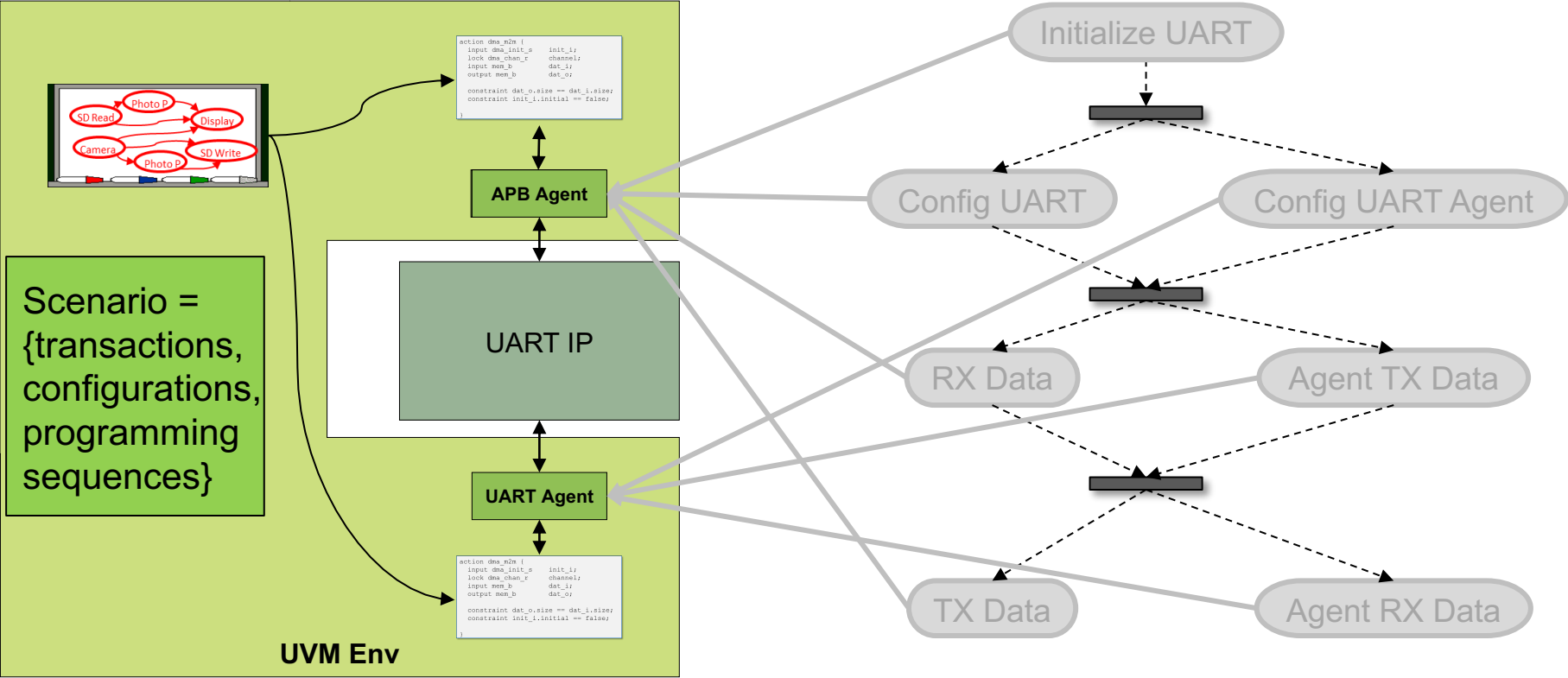
■ Behavior = *Action*



Concise Language to Specify Verification Intent

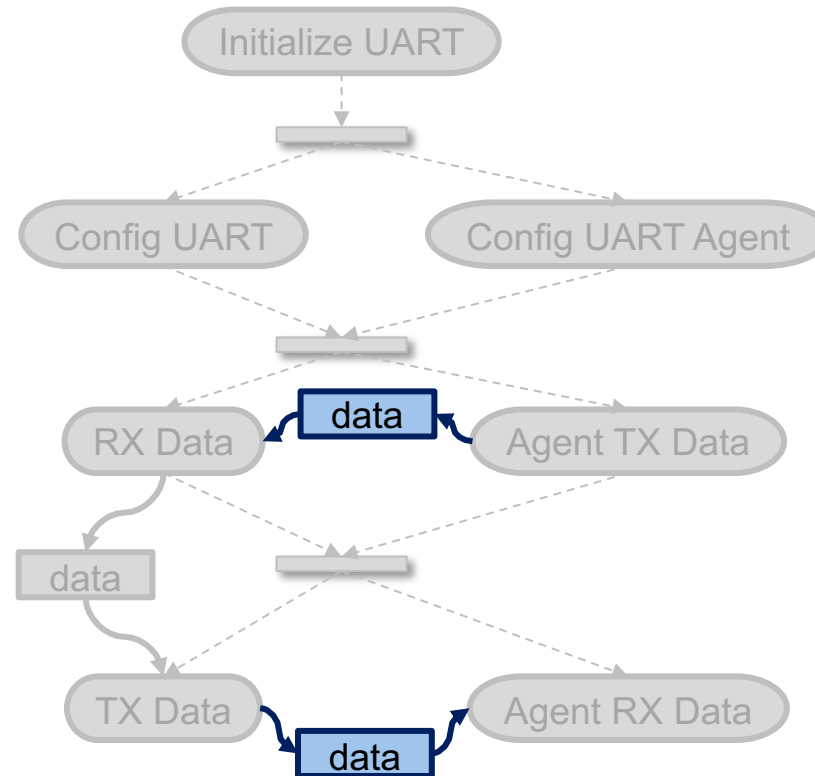
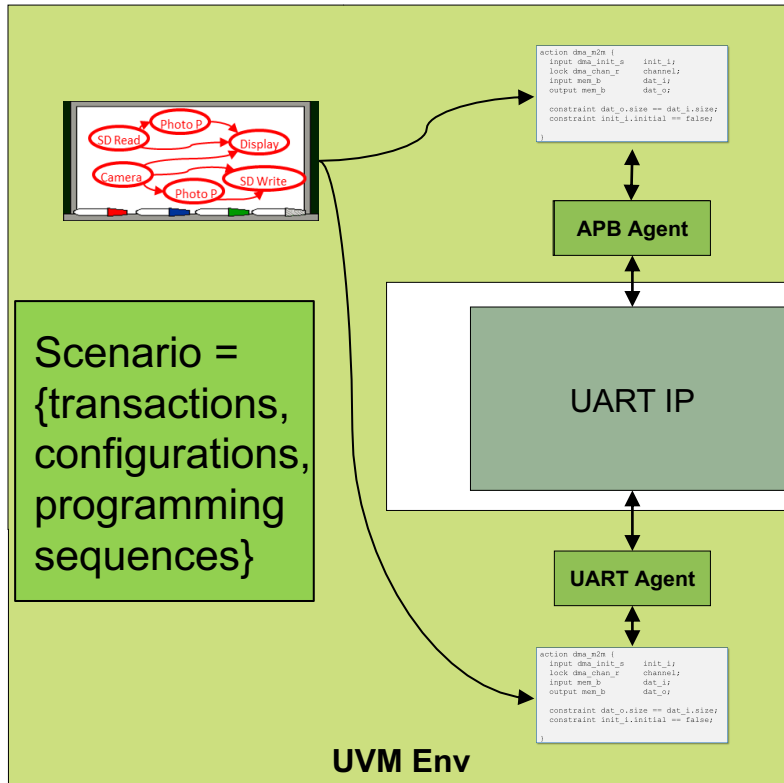
A complement to UVM, not a replacement

- Behavior = *Action*
- Schedule = *Activity*



Concise Language to Specify Verification Intent

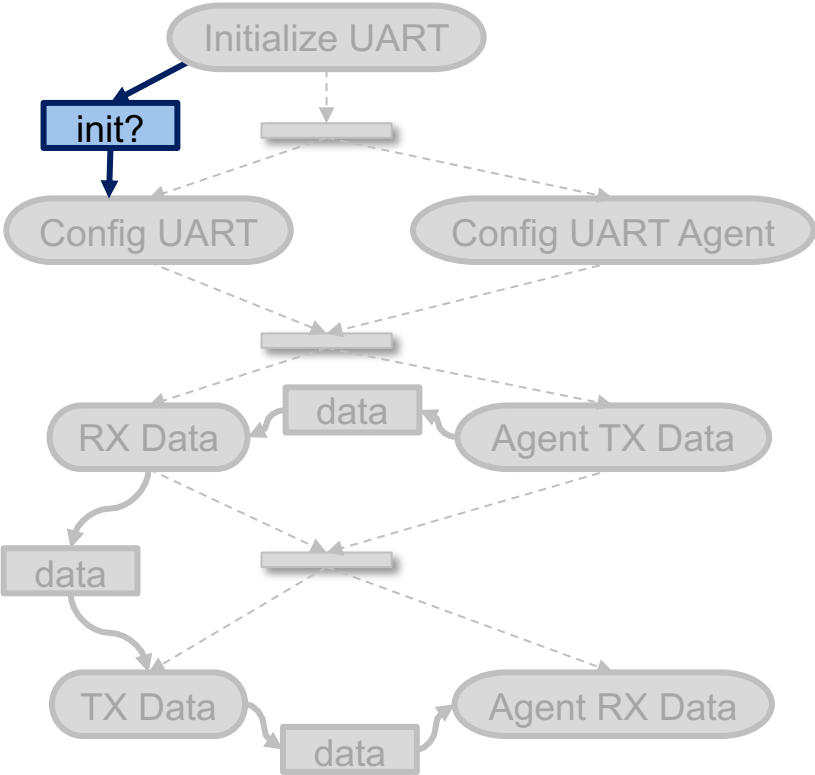
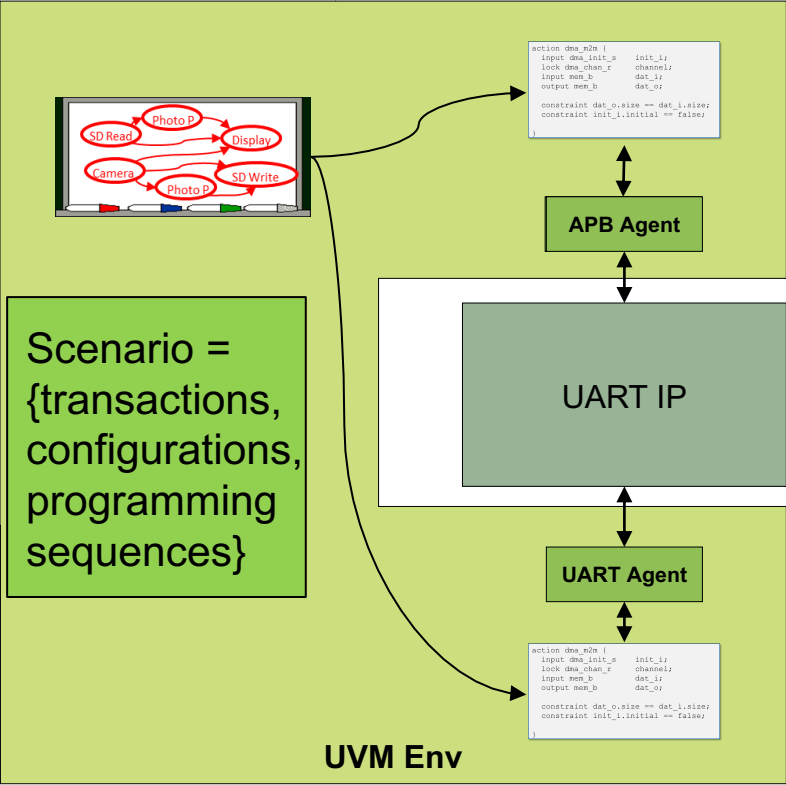
A complement to UVM, not a replacement



- Behavior = *Action*
- Schedule = *Activity*
- Sequential Data = *Buffer*
- **Parallel Data = *Stream***

Concise Language to Specify Verification Intent

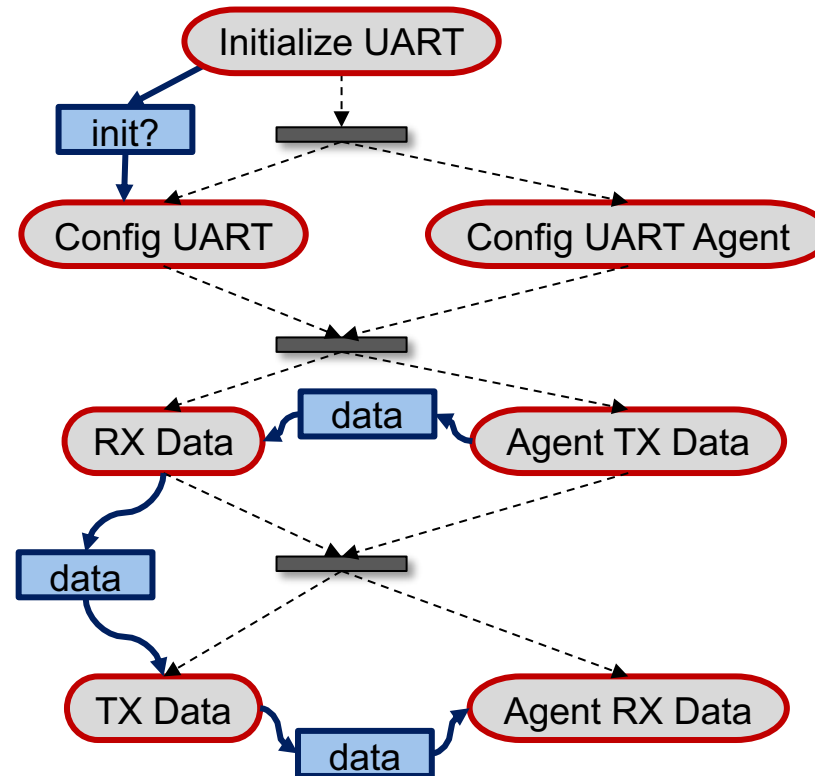
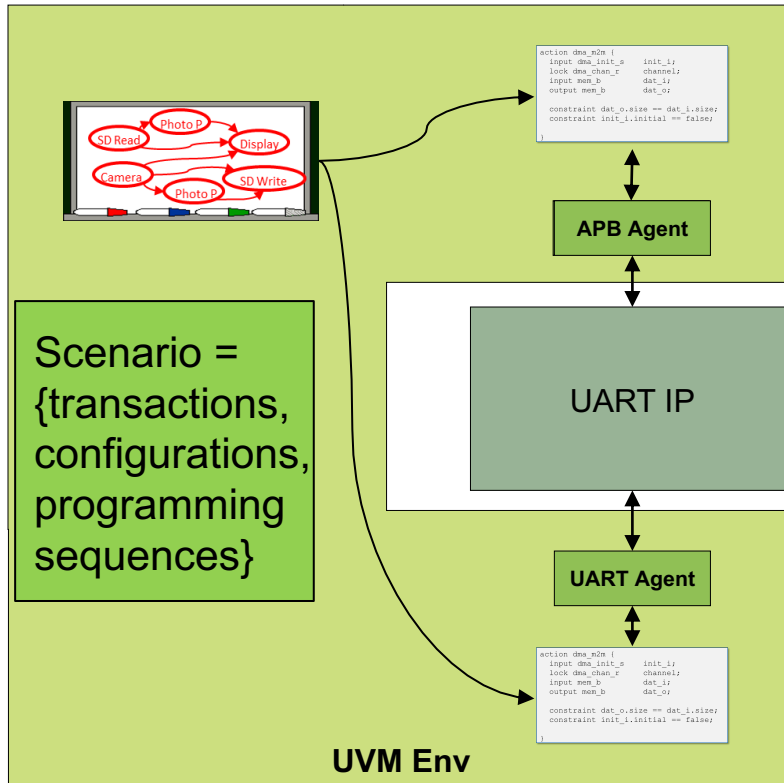
A complement to UVM, not a replacement



- Behavior = *Action*
- Schedule = *Activity*
- Sequential Data = *Buffer*
- Parallel Data = *Stream*
- State info = *State*

Concise Language to Specify Verification Intent

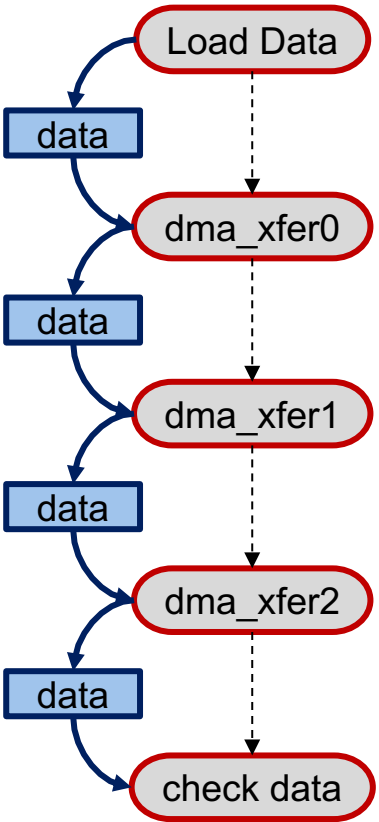
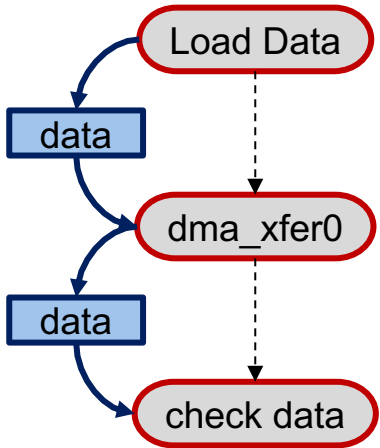
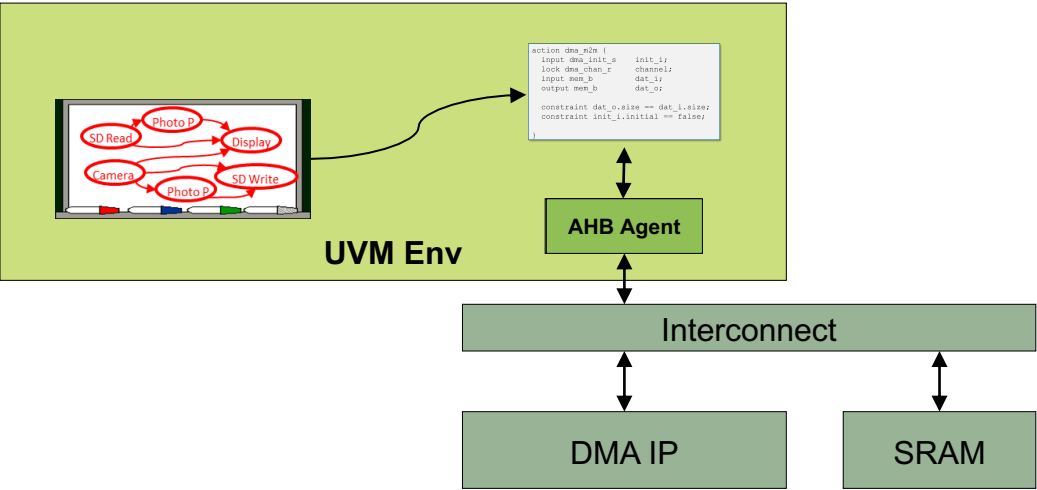
A complement to UVM, not a replacement



- Behavior = *Action*
- Schedule = *Activity*
- Sequential Data = *Buffer*
- Parallel Data = *Stream*
- State info = *State*
- In UVM, PSS can create a set of sequences
 - Run in existing UVM env

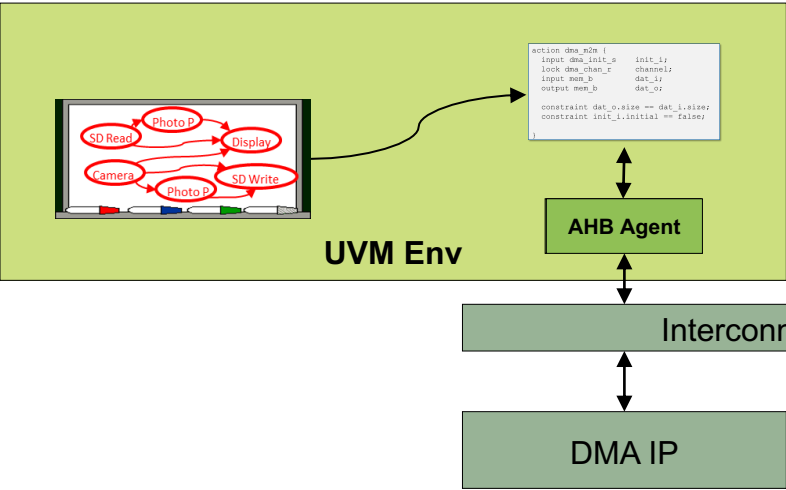
Concise Language to Specify Verification Intent

A complement to UVM, not a replacement



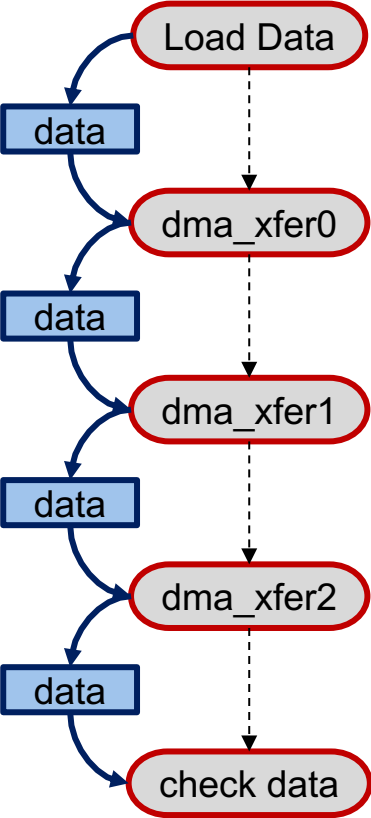
Concise Language to Specify Verification Intent

Easier to specify constraints at scenario level



Scenario constraints:
dma_xfer0.src == load.dest;
check.src == dma_xfer2.dest;
...

Object constraints:
size in [16..128];

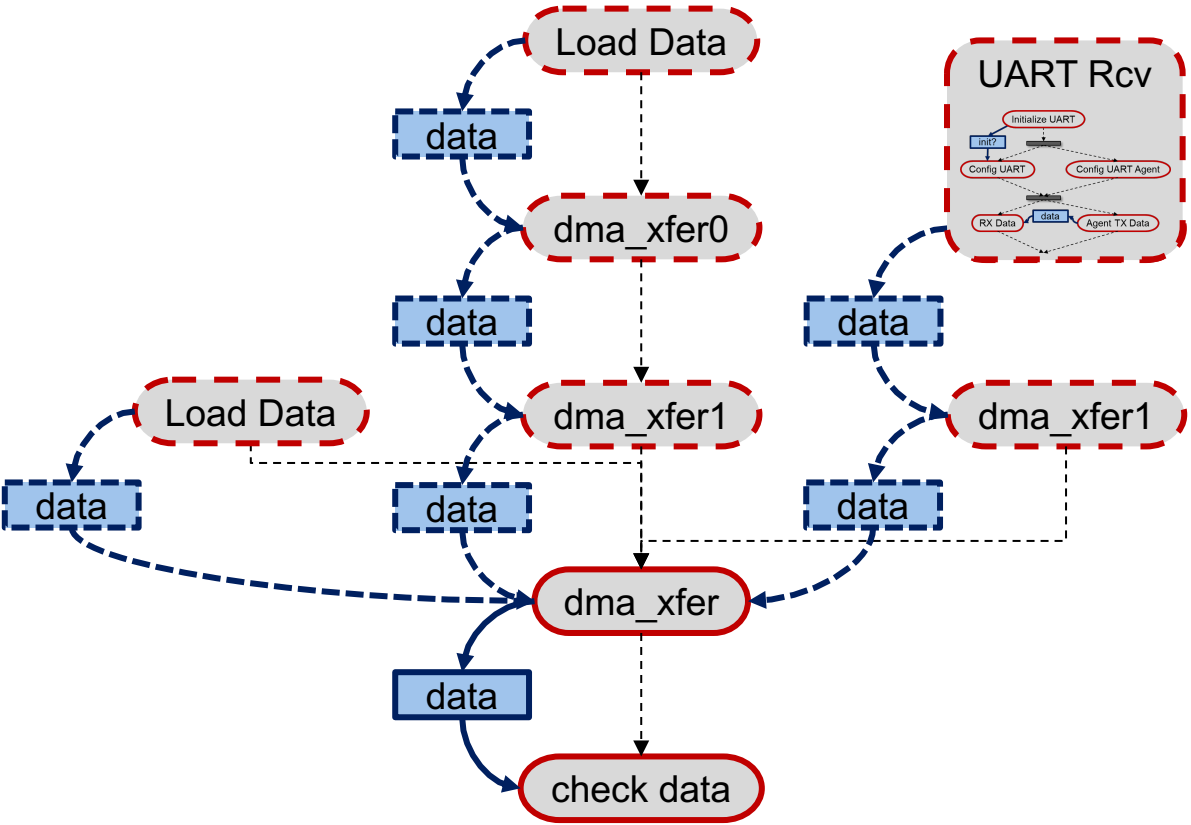
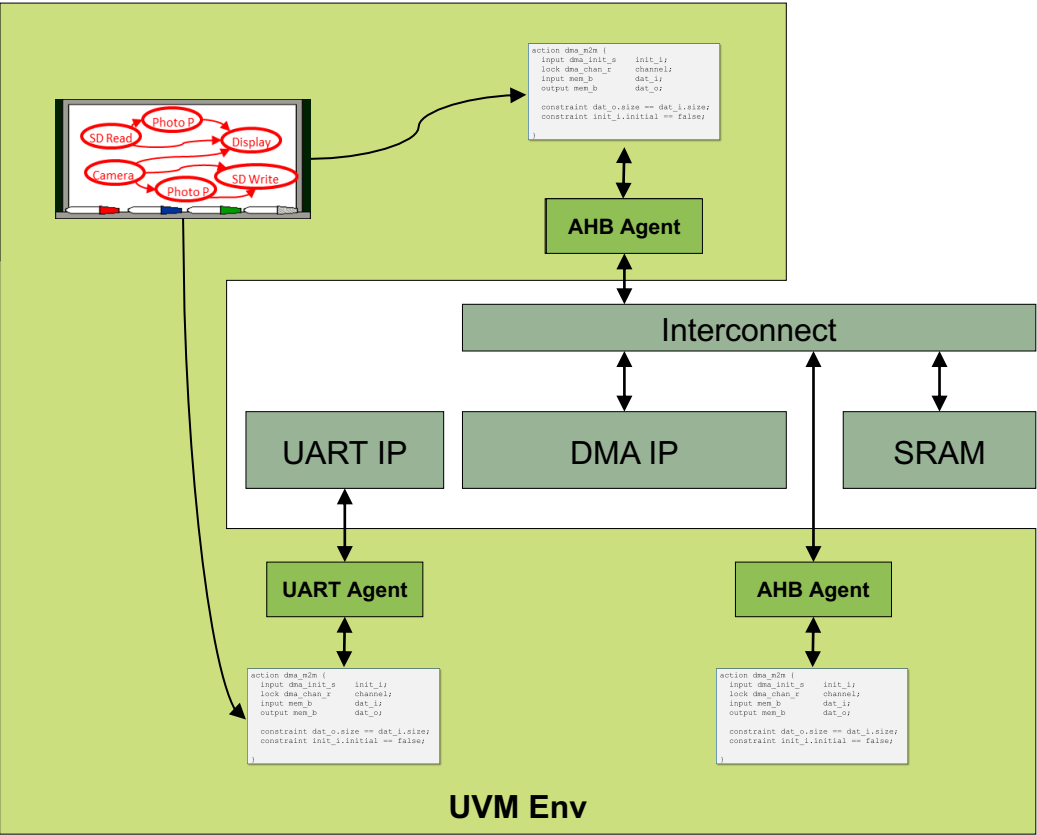


Action constraints:
src.size == dest.size;
src.size <= 64;

Constraints are composable
All relevant constraints are applied across the activity
Can be considered rules for scenario generation

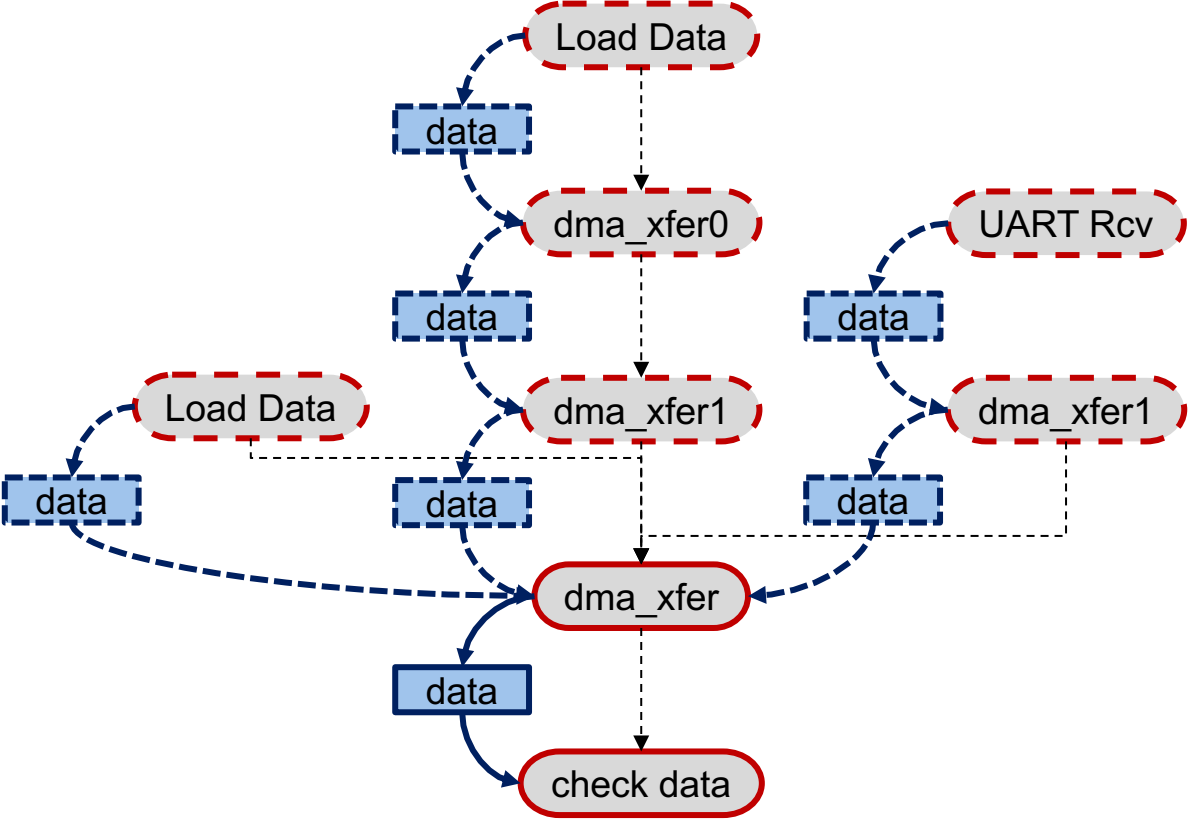
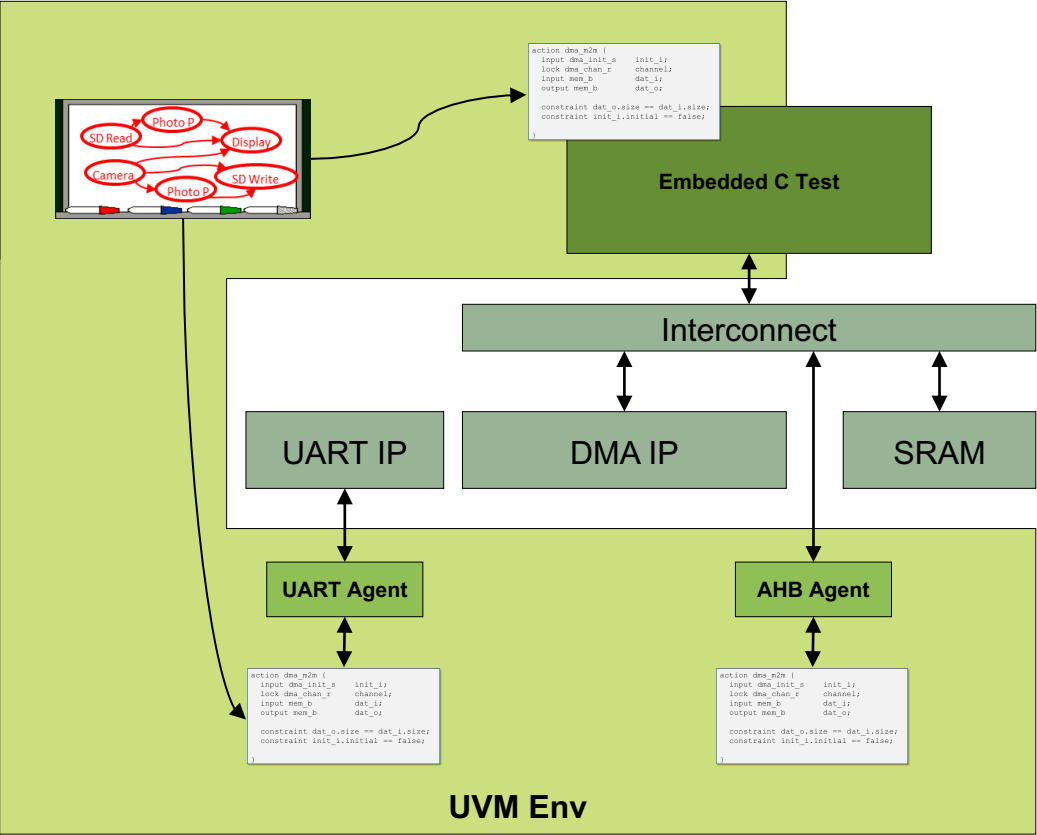
Concise Language to Specify Verification Intent

Rules allow scenarios to be inferred from partial specification



Concise Language to Specify Verification Intent

Scheduling built into generated test regardless of target



The Rubber Meets the Road



The Abstract Model must be implemented on different targets



Atomic Actions → target code

- Target code modeled in exec blocks



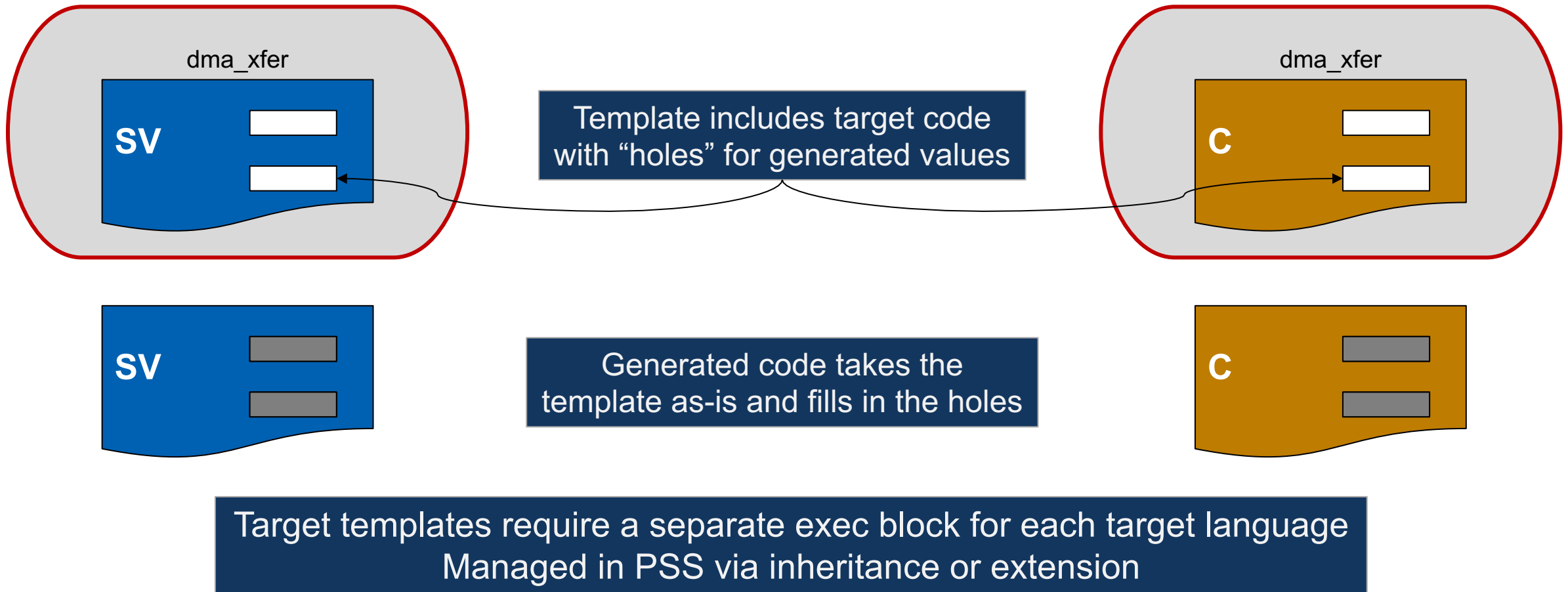
Generator assembles target code according to *Activity* schedule

Action

Exec Block

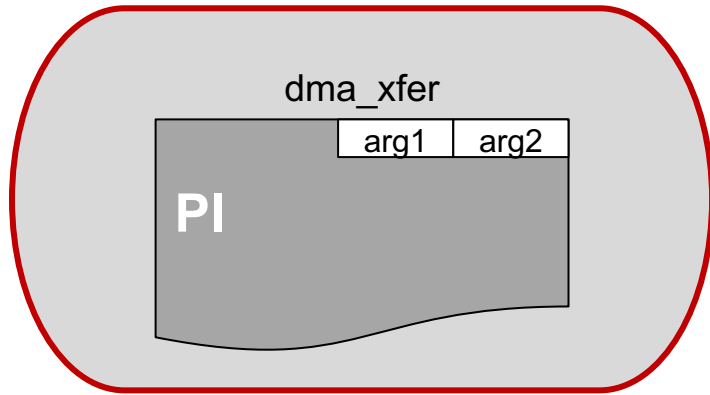
Exec Blocks Define the Target Implementation

Target Templates Define 1:1 Mapping

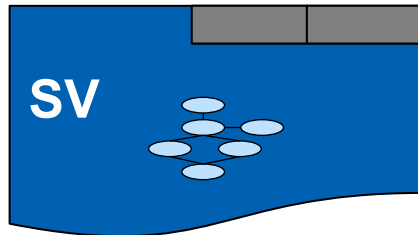


Exec Blocks Define the Target Implementation

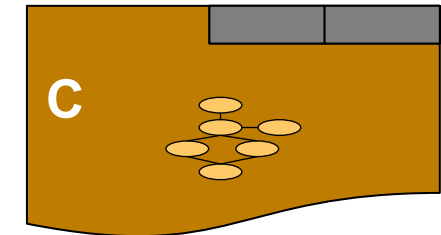
Procedural Interface Isolates Exec Block from Target Language



Procedural Interface defines functions that map to functions(/tasks) in the target language.
Values passed as arguments



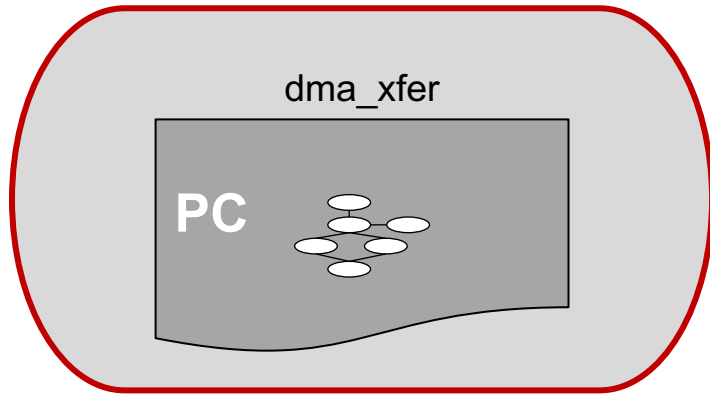
Still have to define possibly complex algorithms in each target language



Procedural Interface lets you have one exec block per action type
Simplifies PSS code management

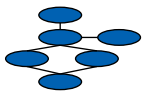
Exec Blocks Define the Target Implementation

Procedural Constructs Move Complex Flow Control to PSS



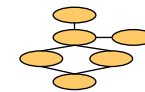
Algorithm is specified in exec block
Imported methods called accordingly

SV



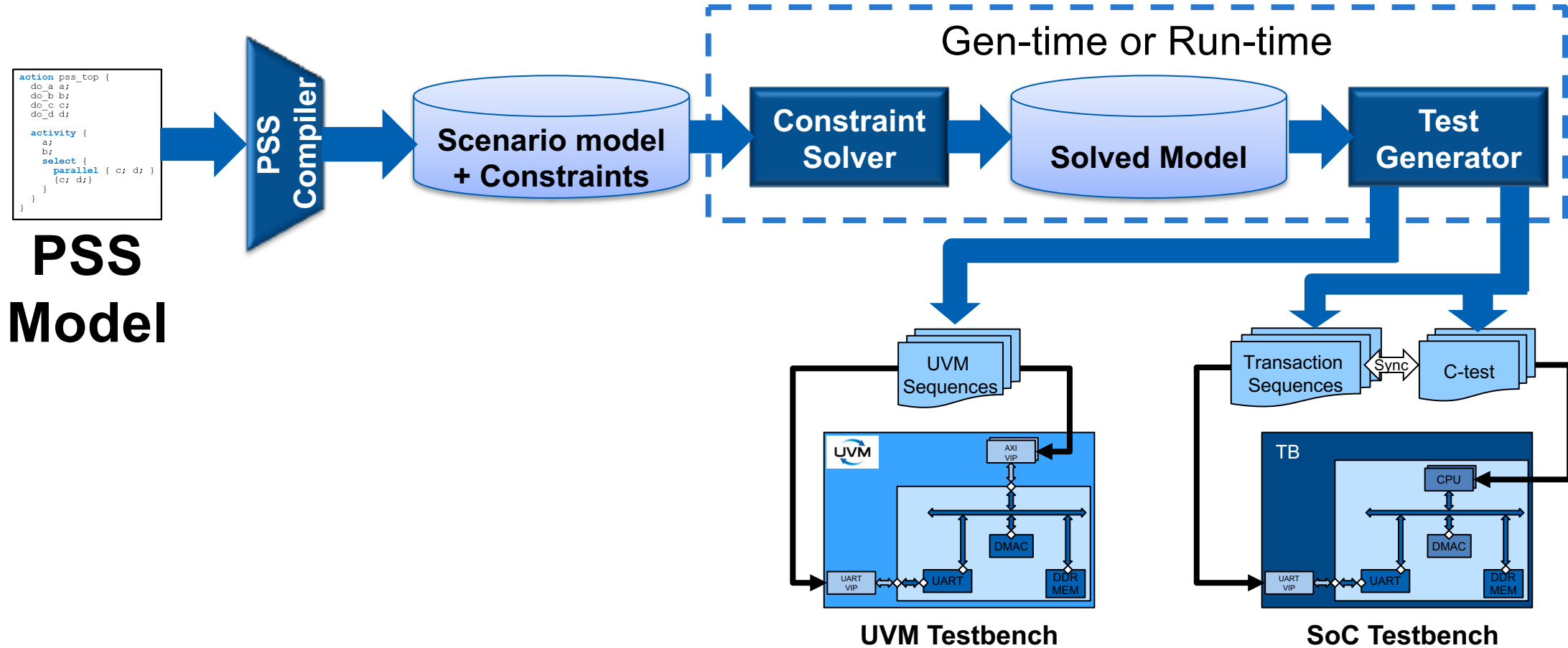
Complex control flow generated from PSS
Language-specific code is much simpler

C

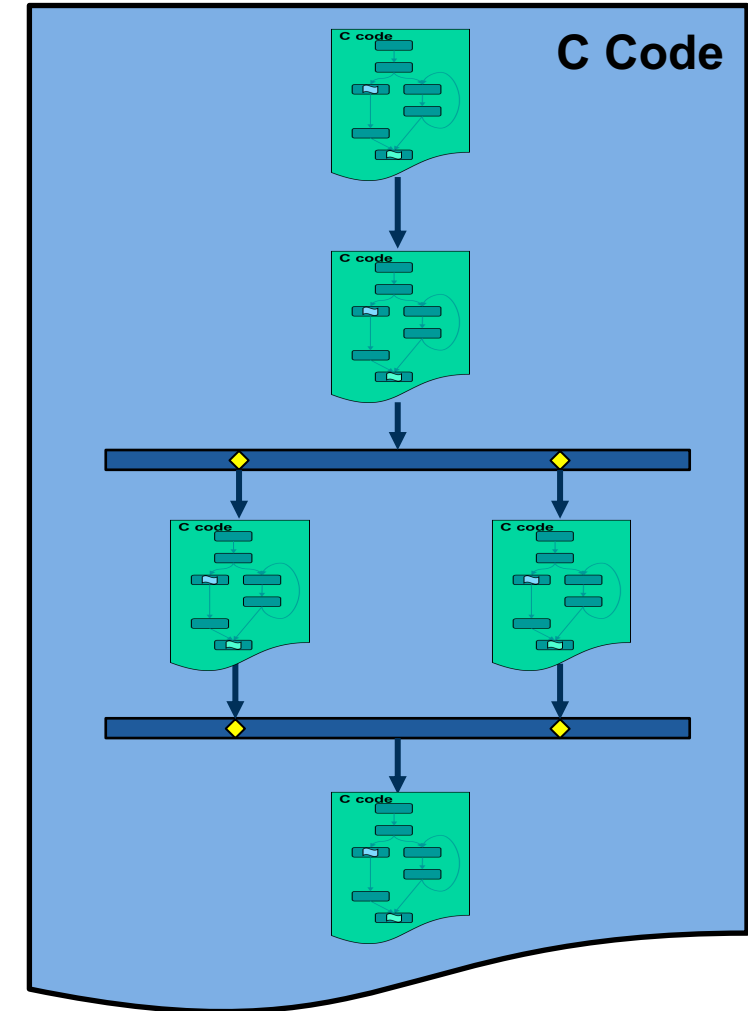
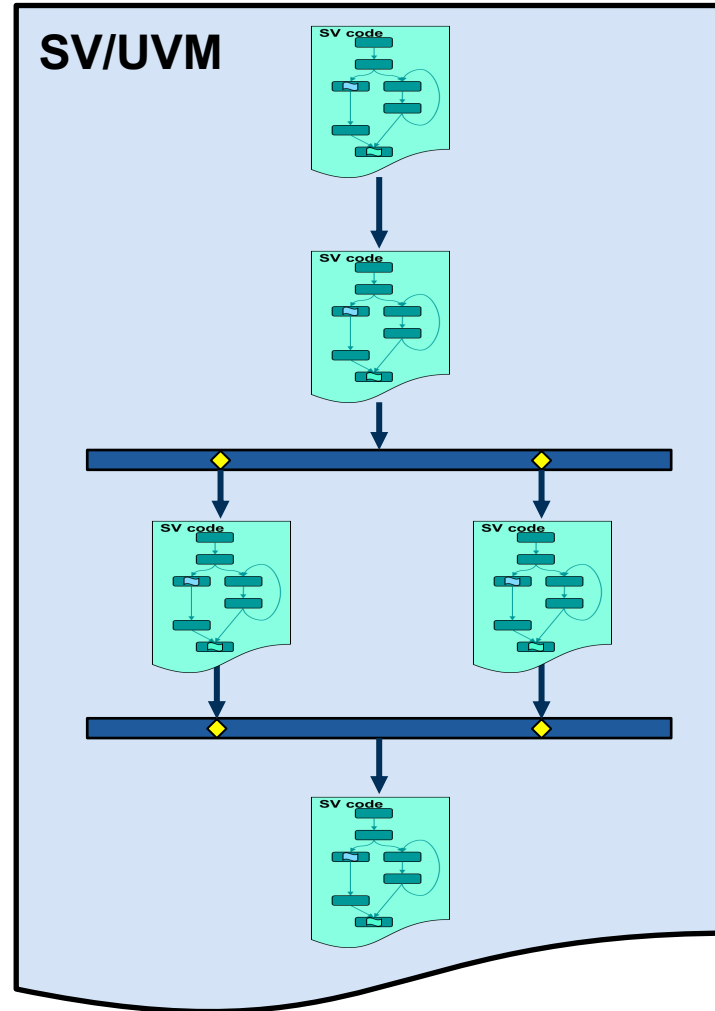
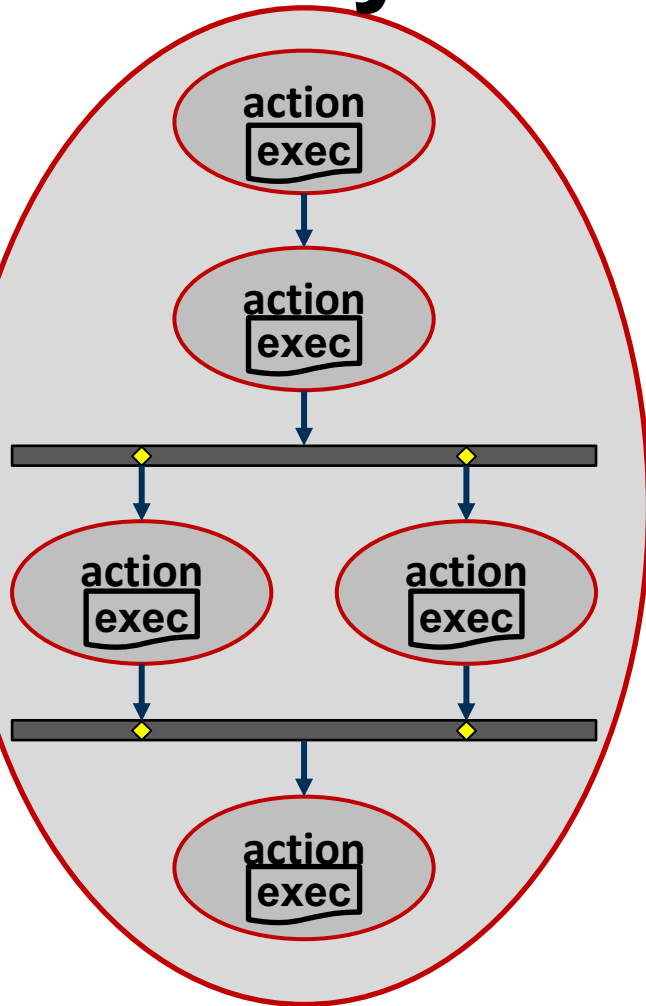


Procedural Constructs provide maximum reuse by
simplifying the migration between languages

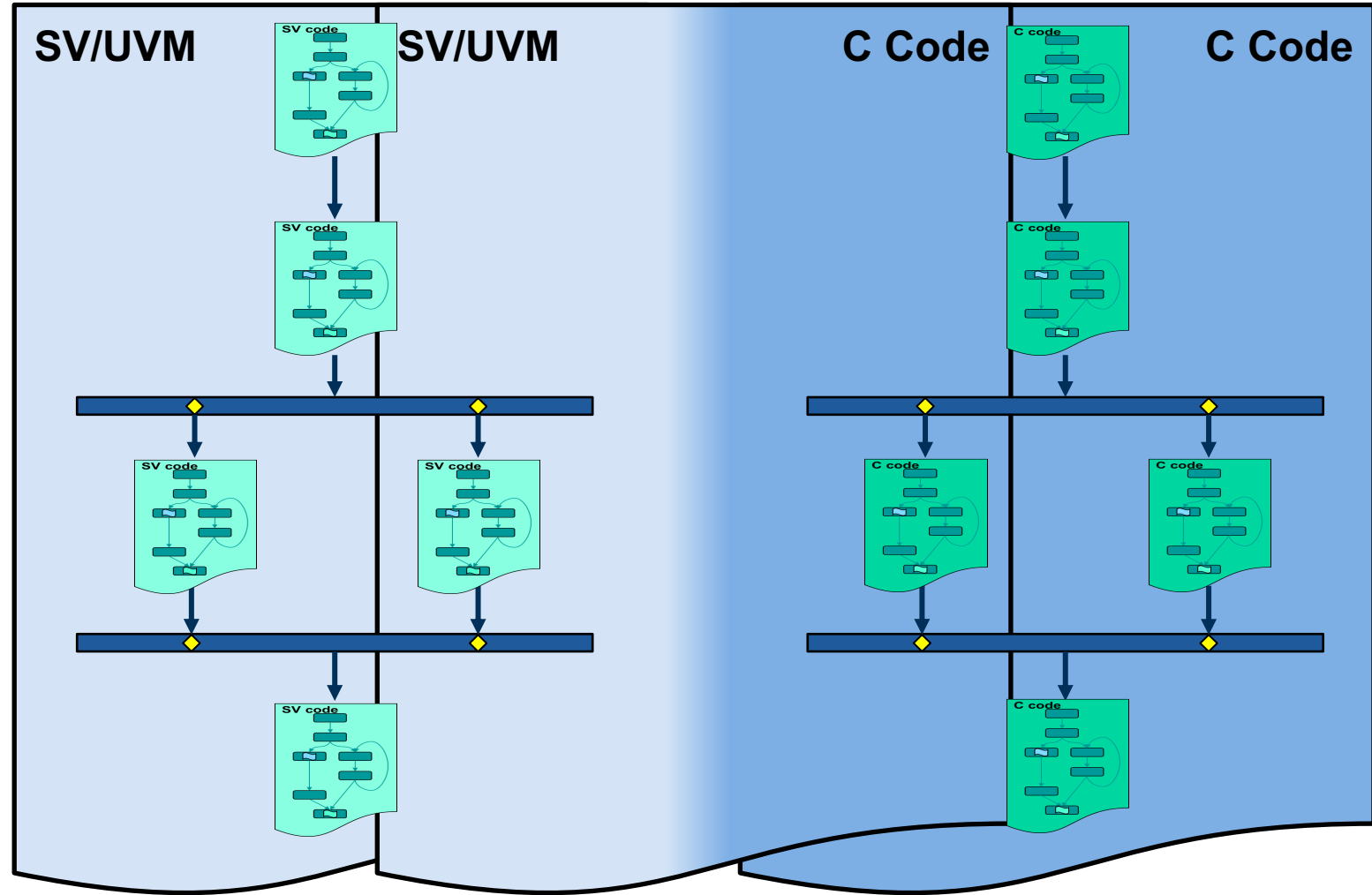
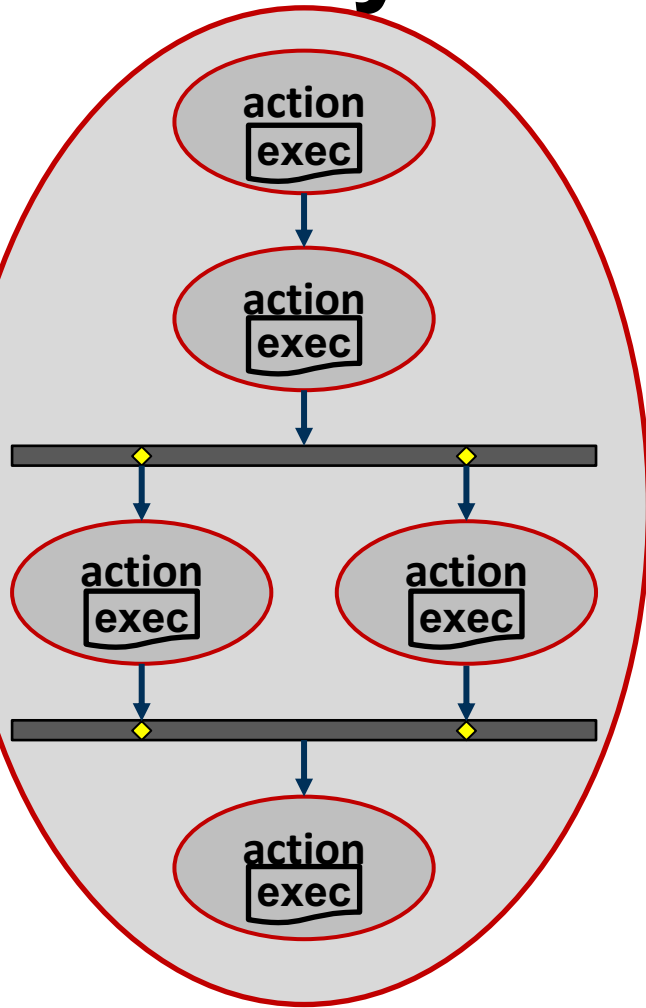
PSS Generalized Tool Flow



Generated Code Assembled According to Activity Schedule



Generated Code Assembled According to Activity Schedule





Developing Reusable Test Content at Block Level

PSS Test Content at Block-IP level

Goals and Requirements

- Create reusable content for IP consumer teams to use
 - Initialize IP in specific modes
 - Exercise key IP operations
- Exercise key configurations as requested by consumer team
 - Collect coverage metrics to confirm
- Test content must run in UVM and embedded-software environments

Benefits to the Block-level Testing

- More-easily create complex scenario-level tests
- Shared medium to discuss test scenarios with other teams
 - Architecture, SoC DV, firmware, driver, validation, etc

PSS Modeling and Realization

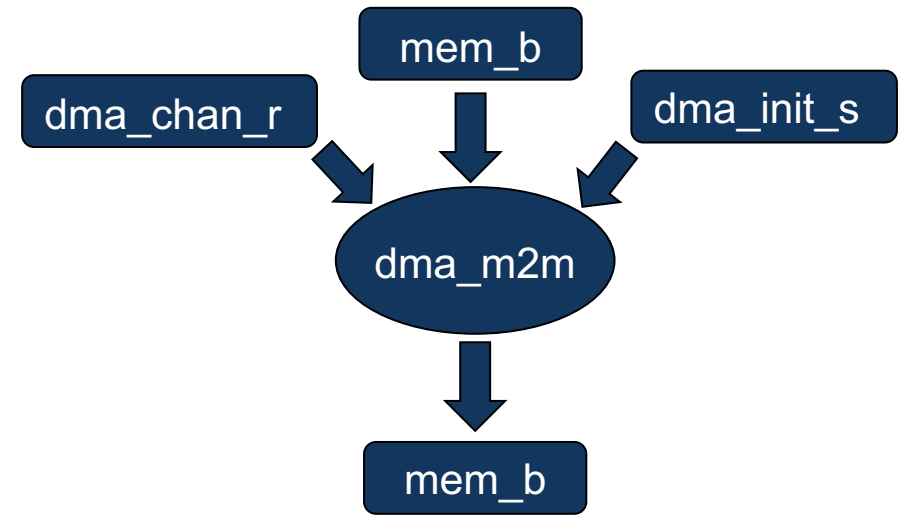
■ Modeling

- Capture the 'what' of a test
- Capture relationships and requirements

■ Realization

- How do we carry out behavior?
- What functions do we call?
- What values (from the modeling layer) do we pass?

■ Data selected in the Modeling layer used in Realization



```
cregs.src_addr.write_val(addr_value(dat_i.mem_h));
cregs.dst_addr.write_val(addr_value(dat_o.mem_h));
cregs.sz.write_field("TOT_SZ", dat_i.size);

cregs.status.write_field("EN", 1);

while (cregs.status.read().DONE == 0) {
    yield;
}
```

Simplifying Register Programming with a RAL

- **Register Access Layers exist to simplify reading/writing registers and avoid mistakes**
 - Define mnemonics for registers and fields, so we don't have to remember addresses and bit positions

```
void init_uart(char *regs, char bits,  
              char stop, char pen, char pev) {  
    regs[3] = ((bits-5) | stop << 2 |  
              pen << 3 | pev << 4);  
}
```

```
void init_uart(uart_regs *regs, char bits,  
              char stop, char pen, char pev) {  
    uart_lcr_reg val;  
    val.bits = (bits-5);  
    val.stop = stop;  
    val.parity_en = pen;  
    val.parity_even = pev;  
    regs->lcr = val;  
}
```

- **Most methodologies have one or more RAL**
 - C/C++ -- structs, unions, macros
 - UVM – UVM register model
 - PSS – PSS register-access layer
- **Most device-specific RALs are generated from a higher-level description**

PSS RAL Overview

- PSS defines data types for capturing a RAL in the *Core Library*
- The PSS RAL targets the requirements of bare-metal software tests
 - Light-weight, intended to enable tools to scale to huge register maps
- The RAL for an IP is easily reused in a larger system context
 - Self-contained and addressed relative to its parent
- Provides access methods that simplify programming-sequence creation
 - Read/write by integer value
 - Read/write by bitfield view
 - Read/modify/write operation to update fields with compact code



- ▼ 23. PSS core library
 - 23.1 String formatting and output
 - 23.2 File operations
 - 23.3 Error reporting
 - 23.4 Randomization
 - 23.5 Floating-point
 - 23.6 Executors
 - 23.7 Address spaces
 - 23.8 Allocation within address spaces
 - 23.9 Data layout and access operations
 - 23.10 Registers

Defining PSS Register Layout

- **PSS *packed struct* specifies register field layout**
 - Specify width of each field
 - Position specifies the offset within the register
- **PSS *register group* collects registers**
 - Contains *reg* fields defined in terms of packed structs
 - Implements a function to map fields to relative offsets
- **PSS RAL types are defined in the PSS Core Library**

```
struct uart_ctrl_ua_mr_reg_s : packed_s<> {  
    bit[1] cclk;  
    bit[2] chr1;  
    bit[3] par;  
    bit[2] nbstop;  
    bit[2] chmode;  
    bit[1] clks;  
    bit[1] irmode;  
    bit[20] reserved;  
}
```

```
pure component uart_ctrl_regs : reg_group_c {  
    reg_c<uart_ctrl_ua_cr_reg_s, READWRITE, 32> ua_cr;  
    reg_c<uart_ctrl_ua_mr_reg_s, READWRITE, 32> ua_mr;  
  
    pure function bit[64] get_offset_of_instance(string name) {  
        if (name == "ua_cr") return 0x0;  
        if (name == "ua_mr") return 0x4;  
    }  
}
```

Reading/Writing Registers with the PSS RAL

- PSS registers provide several read/write APIs

Register Access Function	Purpose
void write(packed_s reg_struct)	Write register struct
packed_s read()	Read register struct
void write_val(bit[SZ] reg_value)	Write register value
bit[SZ] read_val()	Read register value

```
rand bit[2] stop_bits;
constraint stop_bits == 2;

exec body {
    ua_mr_reg_s mr_reg_temp;

    mr_reg_temp.par = 1;
    mr_reg_temp.nbstop = stop_bits;
    mr_reg_temp.chmode = 0;

    // Write Mode Register
    comp.reg.s.ua_mr_reg.write(mr_reg_temp);
}
```

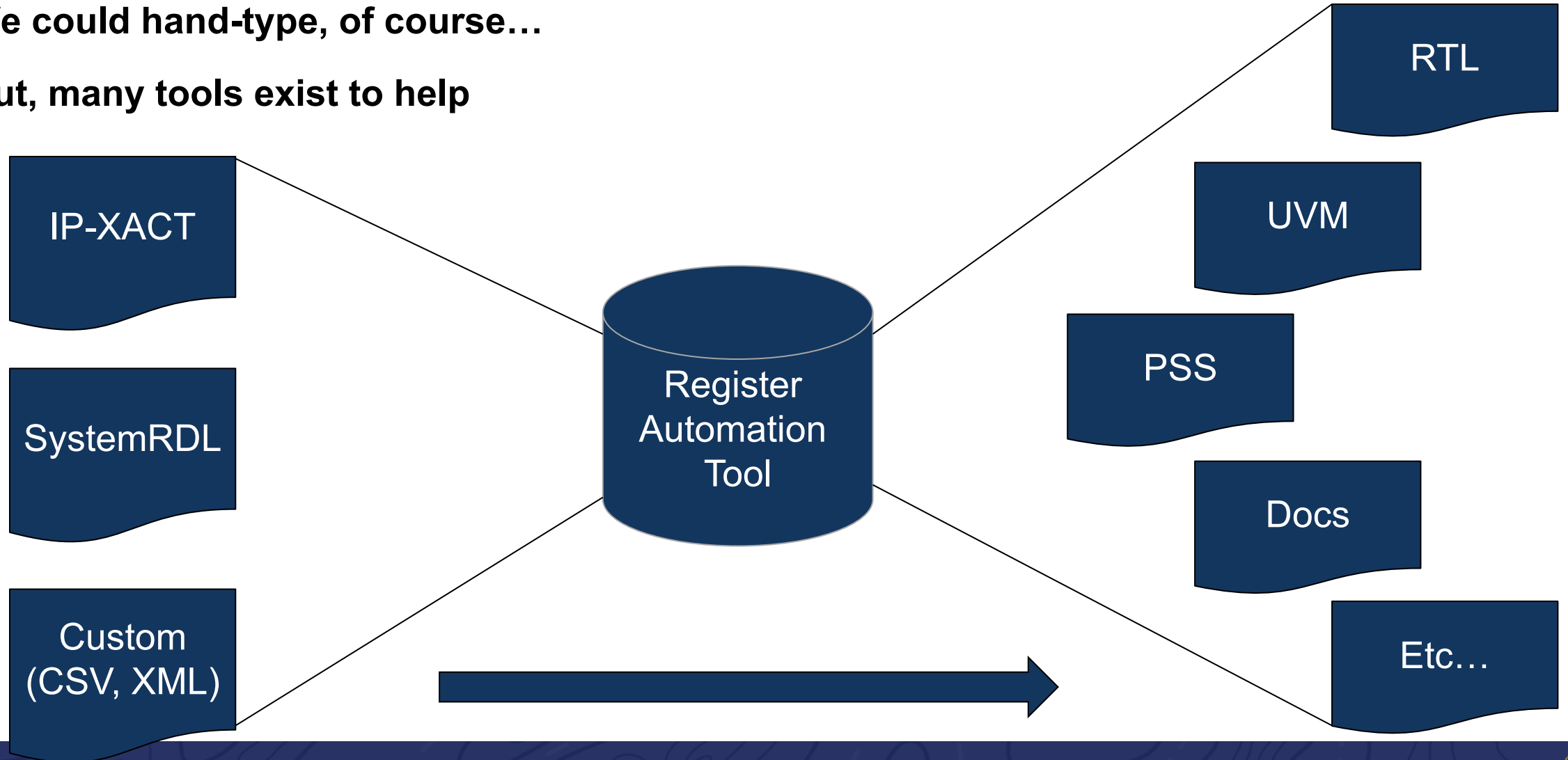
- Single-call read-modify-write simplifies programming sequences

Register Access Function	Purpose
void write_masked(R mask, R val)	Masked write of a struct
void write_val_masked(bit[SZ] mask, bis[SZ] val)	Masked write of a integer
void write_field(bit[string name, bit[SZ] val)	Write a named field
void write_fields(list<string> names, list<bit[SZ]> vals)	Write a set of named fields

```
rand bit[4] mode;
rand bit[16] coeff;
exec body {
    comp.reg.s.cr.write_fields(
        {"mode", "coeff"}, // field names
        {mode, coeff});    // values
}
```

How is a PSS RAL Created?

- We could hand-type, of course...
- But, many tools exist to help

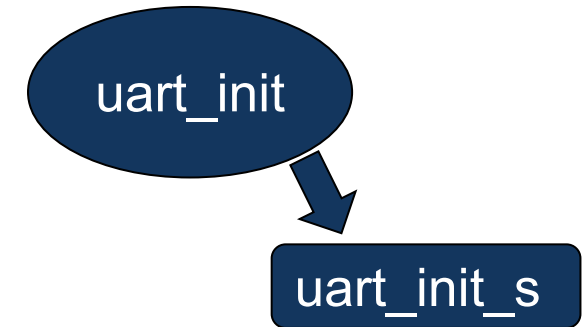


Modeling: IP Behavior - Initialization

- Nearly all IPs need to be initialized before use
 - DMA needs to configure channels and interrupts
 - UART needs to set baud rate, etc
- Typical to use a *state* object to store initialization data
 - Accessible by any action using the IP *component*
 - Prevents changes to the initialized mode while the IP is in use
- Want to provide a variety of initialization actions
 - Full-random initialization
 - Fully-fixed 'sanity' initialization
 - Fixed along specific axes, etc.

```
action uart_init {  
    output uart_init_s    init_o;  
  
    // ...  
}
```

```
state uart_init_s {  
    rand bit[4] in [5..8] bits;  
    rand bit                stop;  
    rand bit                parity_en;  
    rand bit                parity_even;  
    rand bit[32] in  
        [9600,19200,38400,115200] baud;  
}
```



Test Realization: UART Register Definition

- Focus on what we need for initialization
 - Mostly the line-control register (LCR)

Define layout of fields within the register

```
struct lcr_s : packed_s<> {  
    bit[2] bits; // 5=0 ... 8=3  
    bit    stop;  
    bit    parity_en;  
    bit    parity_even;  
    bit[3] rsvd;  
}
```

Specify layout of registers within the UART

```
pure component uart_regs_c : reg_group_c {  
    reg_c<bit[8]>    rxtx_b;  
    reg_c<ier_s>     ier;  
    reg_c<iir_s>     iir;  
    reg_c<lcr_s>     lcr;  
    reg_c<bit[8]>    dlab0;  
    reg_c<bit[8]>    dlab1;  
    // ...  
}
```

Test Realization: Programming Sequence

- Programming sequence is simple and compact

- Translate from selected config mode to registers

```
state uart_init_s {  
  rand bit[4] in [5..8] bits;  
  rand bit      stop;  
  rand bit      parity_en;  
  rand bit      parity_even;  
  rand bit[32] in  
    [9600,19200,38400,115200] baud;  
}
```

Program mode settings

Calculate the divider settings and
program the registers

```
component uart_c {  
  ref uart_regs_c  regs;  
  int SYSCLK_FREQ = 50_000_000;  
  
  action uart_init {  
    output uart_init_s  init_o;  
  
    exec body {  
      // Initialize UART mode  
      comp.regslcr.write({  
        .bits = (init_o.bits-5),  
        .stop = init_o.stop,  
        .parity_en = init_o.parity_en,  
        .parity_even = init_o.parity_even  
      });  
      bit[32] div =  
        comp.SYSCLK_FREQ/(16*init_o.baud);  
      comp.regslab0.write_val(div[7:0]);  
      comp.regslab1.write_val(div[15:8]);  
    }  
  }  
}
```

Modeling: Specialized Initialization Actions

- The base UART initialization action is fully-random
 - Can select any combination of values
- Typically, there is a set of common modes in which to initialize an IP
- Define these as specializations of the base (fully-random) initialization action
 - Some with fully-specified parameters
 - Others with some variability

```
action uart_init_sanity : uart_init {  
  constraint init_o.baud == 9600;  
  constraint init_o.parity_en == 0;  
  constraint init_o.bits == 8;  
  constraint init_o.stop == 1;  
}
```

```
action uart_init_n81 : uart_init {  
  constraint init_o.parity_en == 0;  
  constraint init_o.bits == 8;  
  constraint init_o.stop == 1;  
}
```

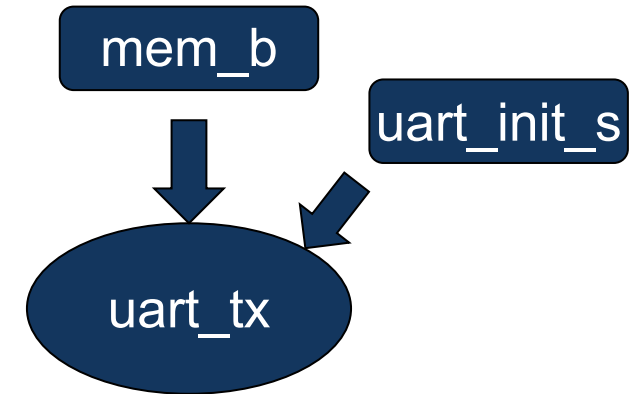
- Automatically reuse register-programming sequence defined in the base action

Modeling: Requiring Initialization

- Encoding pre-conditions is a key aspect of creating reusable test content
 - In this case, that the IP must be initialized, possibly in a specific mode
- PSS *state* objects allow us to require IP initialization before use
 - Initialization actions set the state to non-initial
 - Behavior actions require a non-initial state

Requires at least one initialization action to run before this action

```
action uart_tx {  
  input uart_init_s    init_i;  
  input mem_b          dat_i;  
  constraint !init_i.initial;  
}
```

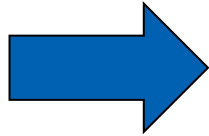


- PSS tools detect if we attempt to use an IP before initialization
 - Randomly *infer* a valid initialization action
 - Report an error if no initialization action exists to be inferred

Methodology: Factoring Out Commonalities

- It's likely that *all* of our behavior actions depend on a properly-initialized IP
- It's good practice to factor out core requirements like this to an abstract base action
 - *Abstract* means that the action is just a building block, and won't independently

```
action uart_tx {  
  input uart_init_s    init_i;  
  input mem_b          dat_i;  
  
  constraint !init_i.initial;  
}
```



```
abstract action uart_base {  
  input uart_init_s    init_i;  
  
  constraint !init_i.initial;  
}
```

```
action uart_tx : uart_base {  
  input mem_b          dat_i;  
  
}
```

...

```
action uart_rx : uart_base {  
  output mem_b         dat_o;  
  
}
```

Placing Requirements on Initialization Mode

- Thus far, we have just required the IP is initialized
 - Any randomly-selected initialization mode is okay
- Often, we also need it to be initialized in some specific way
- Constraining the initialized state adds a requirement
 - Must initialize in high-speed mode to test large data transfer

```
action uart_tx_huge : uart_base {  
    input mem_b          dat_i;  
  
    constraint dat_i.size >= 256*1024;  
    constraint init_i.baud >= 115200;  
}
```

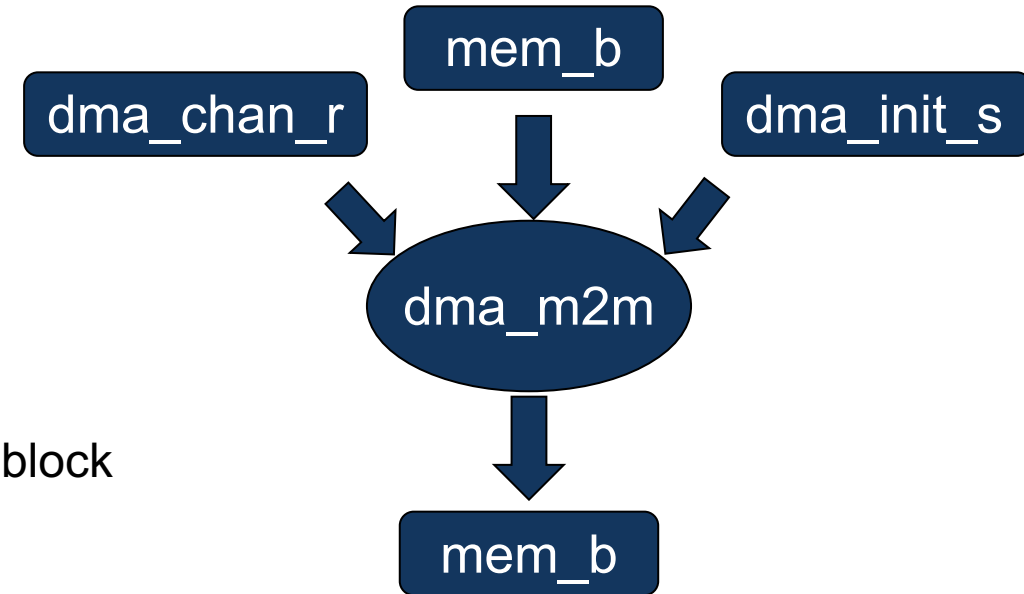
Initialization Coverage - UART

- One of our block-level deliverables is coverage of key initialization modes
- PSS *covergroups* collect coverage metrics
- Covergroups are sampled automatically
 - E.g., at the end of action execution
- Can predict coverage before tests run
 - Coverage is on stimulus – fully under our control
 - Shortens time to identify and close coverage holes

```
action uart_init {  
    output uart_init_s    init_o;  
  
    covergroup {  
        baud_cp : coverpoint init_o.baud;  
        bits_cp : coverpoint init_o.bits;  
        baud_bits_cr : cross baud_cp, bits_cp;  
  
        parity_en_cp : coverpoint init_o.parity_en;  
        parity_even_cp : coverpoint  
            init_o.parity_even  
            iff (init_o.parity_en);  
        parity_cr : cross  
            parity_en_cp,  
            parity_even_cp iff (init_o.parity_en);  
    } uart_init_cov;  
}
```

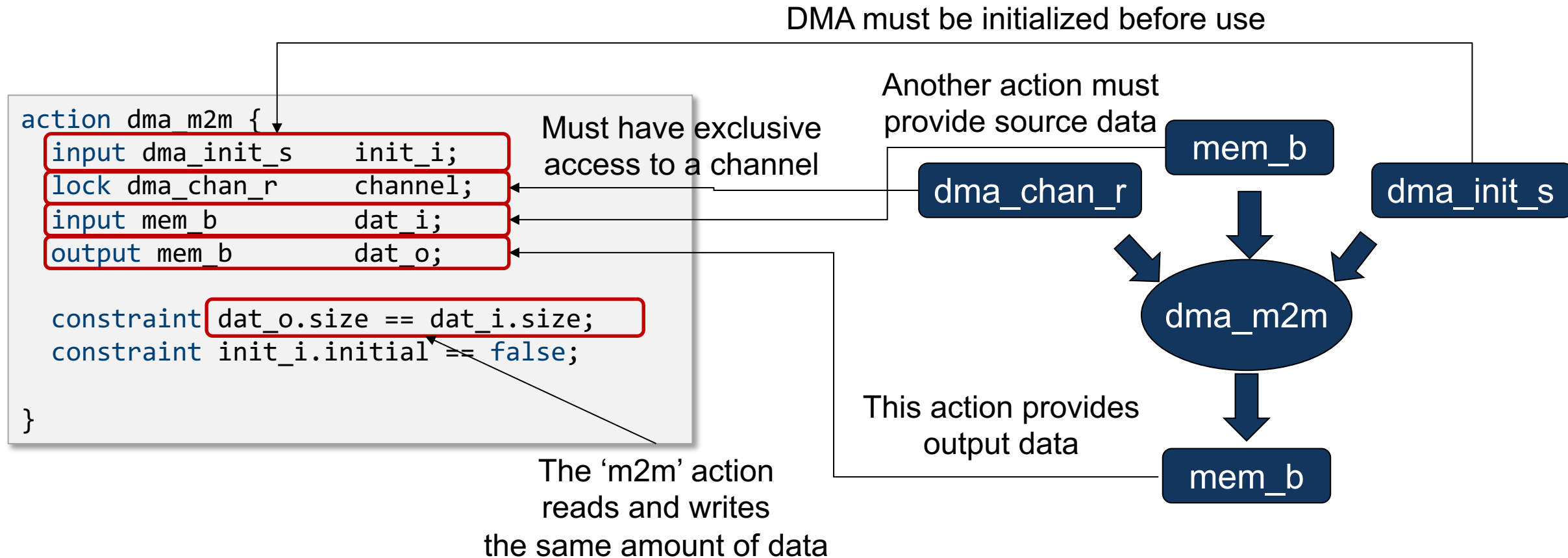
DMA Behavior – Single DMA Transfer

- **Our simplest DMA operation is a memory-to-memory copy**
 - It copies memory from a source memory block to a destination
- **Memory-to-memory pre-conditions**
 - DMA IP must have been initialized
 - Action must have dedicated access to a DMA channel
 - Action must be supplied source memory block to read
- **Memory-to-memory post-conditions**
 - Memory-to-memory operation produces a destination memory block



PSS Action Outline

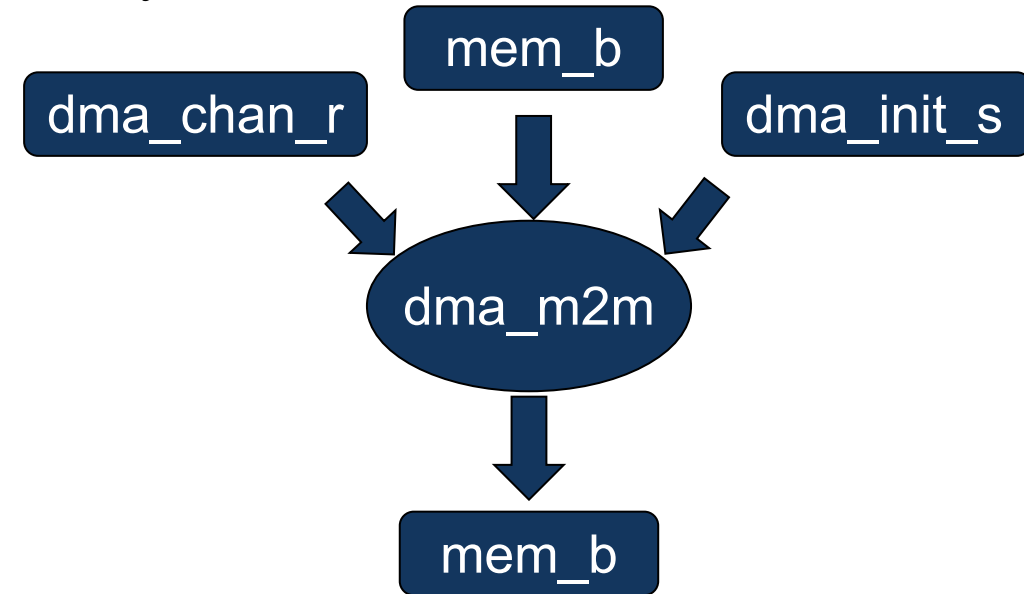
- Our PSS 'memory-to-memory' action captures these requirements
- And, enforces some required relationships



Modeling: Claiming Memory

- The mem-to-mem action produces a block of memory
 - Memory is allocated via a memory *claim* within the memory buffer
 - The newly-created memory block is passed out via the output memory buffer
- Claimed memory is automatically allocated and freed
 - Ensures parallel activity uses unique memory regions
 - Avoids memory leaks

```
buffer mem_b {  
  rand bit[32]      size;  
  addr_handle_t     mem_h;  
  rand addr_claim_s<> mem_claim;  
  
  constraint mem_claim.size == size;  
  exec post_solve {  
    mem_h = make_handle_from_claim(mem_claim);  
  }  
}
```



- Org-common library
- Think about consumer team
- Data producer?
- Accellera-std library in the works

Test Realization – DMA Single Transfer

- We can now implement the connection between action-level model and device registers
- Easily specify programming seq
 - Standard control-flow statements
 - Register- and memory-access methods

Get the channel registers for the target DMA channel

Integer values can be written to registers

Individual fields can be updated with read-modify-write operations

Wait for transfer completion

```
action dma_m2m {  
  input mem_b          dat_i;  
  input dma_init_s      init_i;  
  lock dma_chan_r       channel;  
  output mem_b          dat_o;  
  // ...  
  exec body {  
    ref channel_regs cregs =  
      comp.registers.channels[channel.instance_id];  
  
    cregs.src_addr.write_val(addr_value(dat_i.mem_h));  
    cregs.dst_addr.write_val(addr_value(dat_o.mem_h));  
    cregs.sz.write_field("TOT_SZ", dat_i.size);  
  
    cregs.status.write_field("EN", 1);  
  
    while (cregs.status.read().DONE == 0) {  
      yield;  
    }  
  }  
}
```

Modeling: Encapsulating Complex Behaviors

- **IP operations often involve multiple steps that, as a group**
 - Place internal and external requirements around memory lifetime
 - Place temporal requirements on resource availability
- **PSS enables encapsulating these behaviors with their requirements**
 - Ensures that the behaviors are internally consistent
 - Ensures that usage is consistent with requirements
- **Goal is to deliver easy-to-use behaviors**
 - Expose top-level 'knobs' to enable control
 - Hide details from end users

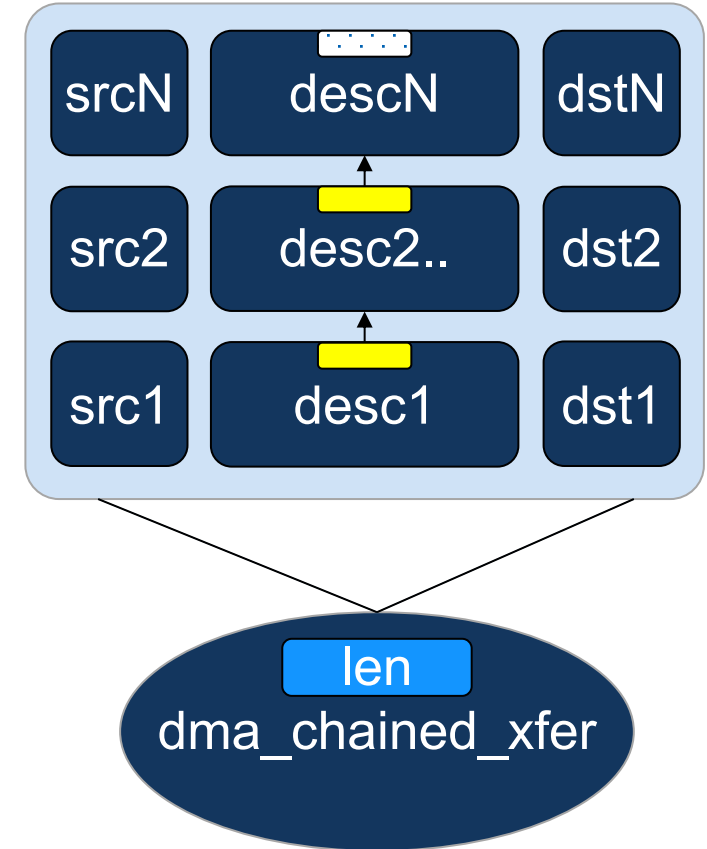
Modeling: Encapsulating Complex Behaviors

- **The DMA Engine supports chained transfers via in-memory descriptors**

- Each descriptor performs a copy between memory regions
- Descriptors 'linked' together into a descriptor chain
- Descriptor-chain memory must be valid for the duration of the transfer
- Data in source regions must be valid before transfer starts
- Data in destination regions is only legal once the full transfer completes

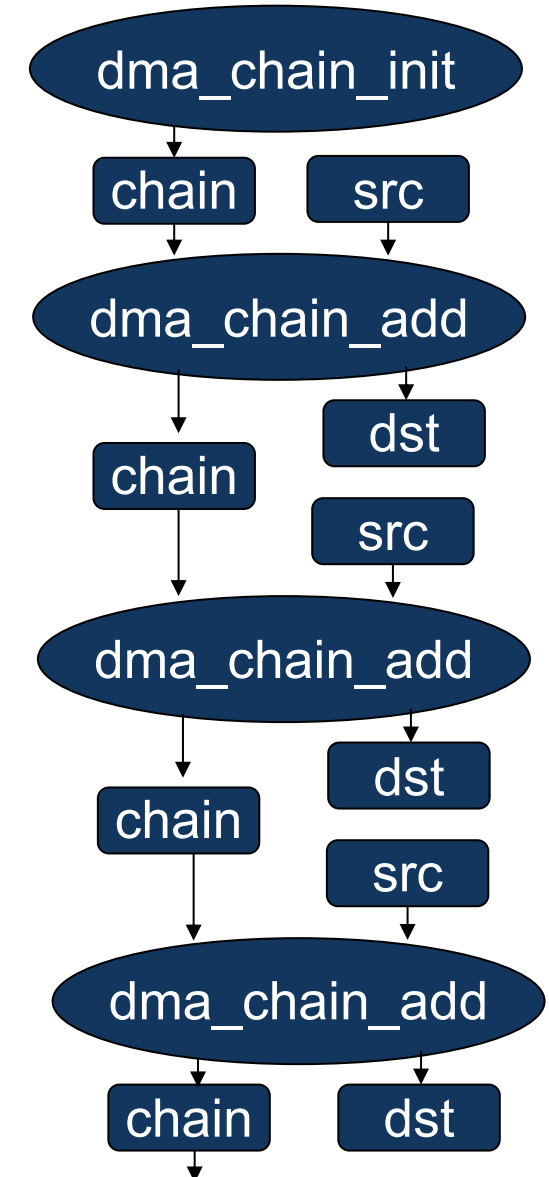
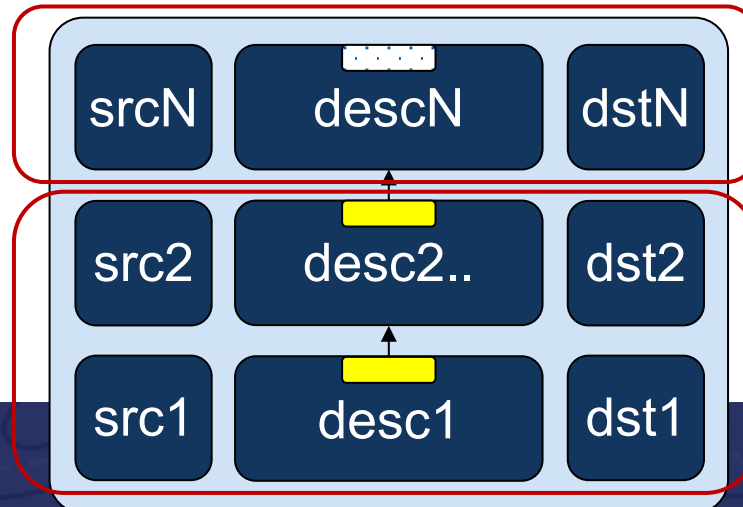
- **Goal: encapsulate task of creating and running a chained transfer**

- Capture requirements around memory usage and lifetime
- Capture programming sequence for descriptor setup and transfer
- Provide a simple action to produce chained transfers of various lengths



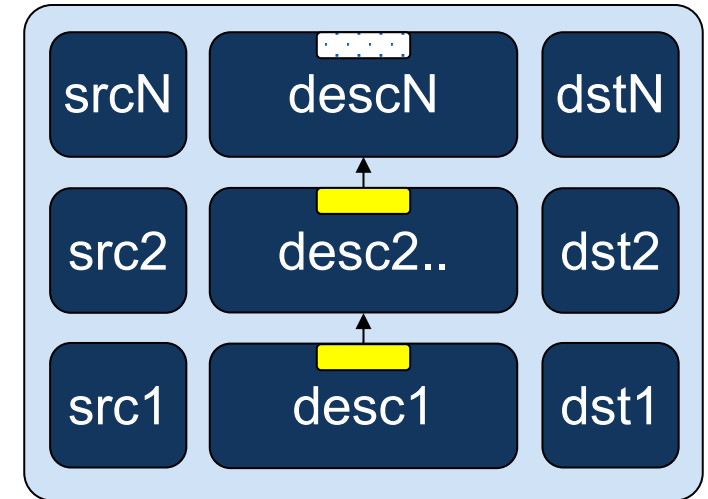
Modeling: Building a Descriptor Chain

- **Key operation: add descriptor to the chain**
- **Two possibilities**
 - Add last descriptor -- 'next' pointer points to null
 - Add non-last descriptor -- 'next' pointer points to previous
- **Key data: last descriptor pointer and accumulated memory blocks**
 - Model with a *buffer*
- **Two actions:**
 - Initialize chain (marks end of the chain)
 - Add new descriptor



Modeling: DMA 'chain' buffer

- **The descriptor chain is built starting at the end**
 - First descriptor built is the tail of the chain
 - Last descriptor built is the head of the chain processed by DMA
- **Must manage two things while building the transfer**
 - Handle to the previously-build descriptor
 - “next” descriptor for the DMA engine to process
 - Handle to memory regions used by the transfer
 - Prevents them from being freed until they’ve been used
- **Encapsulate this data in a *buffer***
 - *list* to hold memory handle
 - *address handle* pointing to the next descriptor
 - Or, null, if at the end of the chain



```
buffer dma_chain_b {  
    list<addr_handle_t>    mem_h;  
    addr_handle_t          next_desc;  
}
```

Modeling: Add-Descriptor Action

- **The add-descriptor action**
 - Claim source, destination, descriptor, memory
 - Propagates the 'chain' buffer data
- **Remember: just setting up the transfer**
 - The full chained transfer runs later

Specify relationships around
allocated memory

Update descriptor-chain 'head'
and 'previous' references

Save all address handles to
extend their lifetime to the end of
the transfer

```
action dma_chain_add {  
  input chain_b      chain_i;  
  output chain_b     chain_o;  
  input dat_b        dat_i;  
  output dat_b       dat_o;  
  rand addr_claim_s<> desc_claim;  
  addr_handle_t      desc_h;
```

```
  constraint desc_claim.size ==  
    sizeof_s<dma_desc_s>::nbytes;  
  constraint dat_i.size == dat_o.size;
```

```
  exec post_solve {  
    desc_h = make_handle_from_claim(desc_claim);  
    chain_o.next_desc = desc_h;
```

```
    chain_o.mem_h = chain_i.mem_h;  
    chain_o.mem_h.push_back(desc_h);  
    chain_o.mem_h.push_back(dat_i.data_h);  
    chain_o.mem_h.push_back(dat_o.data_h);  
  }
```

```
}
```

Test Realization: Descriptor Packed Struct

- Represent in-memory descriptors with packed structs
- Use to model DMA-descriptor memory

Model subfields when needed

```
struct dma_desc_csr_s : packed_s<> {  
    bit[12]      sz;  
    bit[4]       rsvd1;  
    bit          dst, src;  
    bit          inc_dst, inc_src;  
    bit          eol;  
}
```

Combine with fields of other
fixed-size data types

```
struct dma_desc_s : packed_s<> {  
    dma_desc_csr_s  csr;  
    bit[32]         src_addr;  
    bit[32]         dst_addr;  
    bit[32]         next;  
}
```

Test Realization: Populating Descriptor Chain Link

- Populate DMA 'desc'
- Write to memory

Populate transfer size and src/dst addresses

Populate next-descriptor pointer

Write descriptor to memory

```
action dma_chain_add {  
    input chain_b      chain_i;  
    output chain_b     chain_o;  
    input dat_b        dat_i;  
    output dat_b       dat_o;  
    rand addr_claim_s<> desc_claim;  
    addr_handle_t      desc_h;  
  
    // ...  
    exec body {  
        dma_desc_s desc;  
        desc.sz = dat_i.size;  
        desc.src_addr = addr_value(dat_i.data_h);  
        desc.dst_addr = addr_value(dat_o.data_h);  
        desc.csr.eol = (chain_i.next_desc == null)?1:0;  
        desc.next = (chain_i.next_desc == null)?  
            addr_value(chain_i.next_desc):0;  
        write_struct(desc_h, desc);  
    }  
}
```

Test Realization: Running Chained Transfer

- Many similarities to single-transfer
- Start and poll for completion via registers

Get handle to channel-specific registers

Specify the head of the descriptor chain

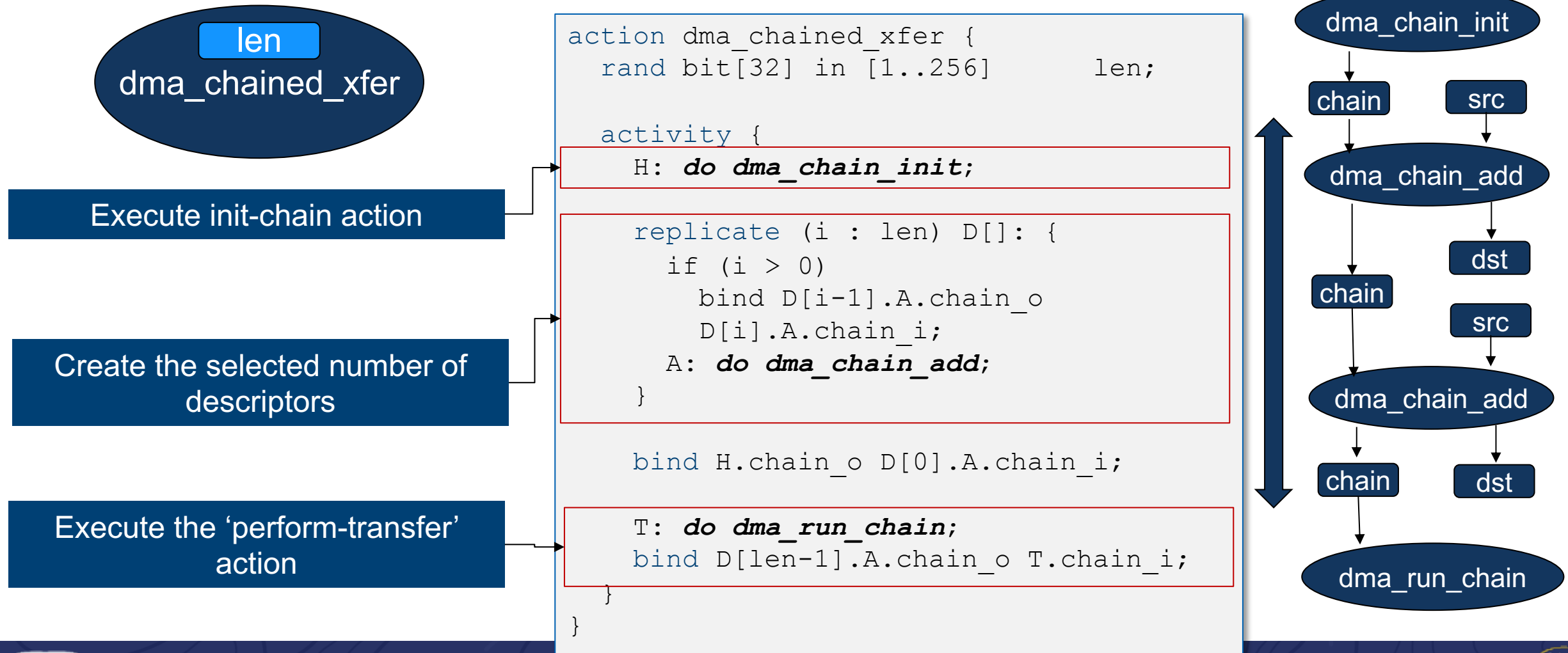
Enable the DMA channel in 'external descriptor' mode

Wait for transfer to complete

```
action dma_run_chain {  
  input chain_b      chain_i;  
  lock dma_chan_r    channel;  
  
  exec body {  
    ref channel_regs cregs =  
      comp.registers.channels[channel.instance_id];  
  
    cregs.desc.write_val(  
      addr_value(chain_i.next_desc));  
  
    cregs.status.write_fields(  
      {"USE_ED", "EN"}, {1, 1});  
  
    while (cregs.status.read().DONE == 0) {  
      yield;  
    }  
  }  
}
```

Modeling: Encapsulating Transfer-Chain Building

- Now, let's create the reusable 'descriptor-chain transfer' action



Modeling Wrap-up: IP-centric PSS Component

- **We've been focused on the IP *behaviors***
 - Modeling pre-conditions and requirements
 - Modeling test realization targeting memory and registers
- **We encapsulate those behaviors (actions) with required IP resources in a *component***
 - Reference to the IP register group
 - Pool of resources
 - *state* pool that holds the current initialized state
- **IP component is independent of integration level**
 - Same at IP, subsystem, and SoC
- **Environment-specific details go in the containing component**

```
component dma_c {  
  ref dma_regs_c      regs;  
  dma_chan_r [4]      channels;  
  pool dma_init_s      init_s;  
  
  action dma_m2m { /* ... */ }  
}
```

PSS Environment Integration

- **Every verification environment has specific characteristics**
 - Memory map
 - Mechanism used to access memory
 - ...
- **Collect these specifics in a top-level PSS *component***
 - Ensures that IP-specific component is environment-independent

IP-specific PSS component

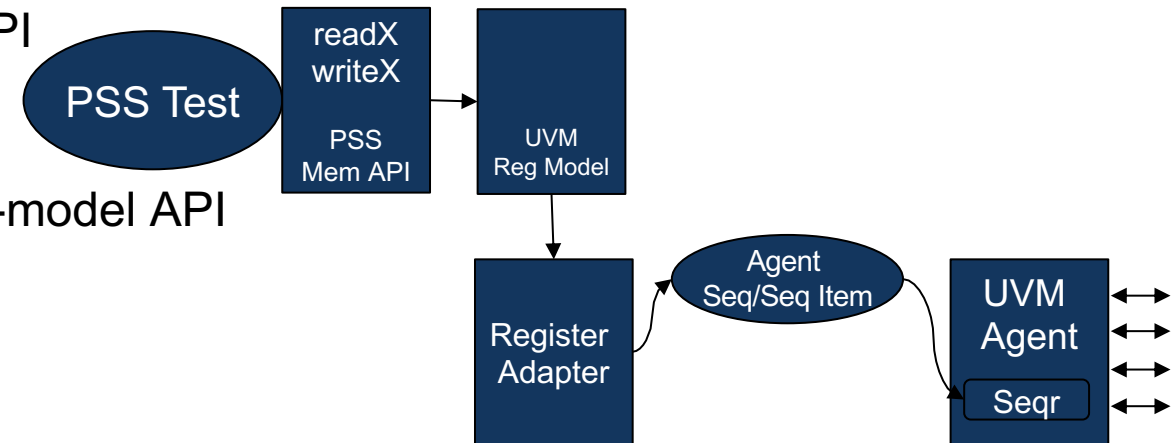
Env-specific register block

Connecting IP register reference
to env-specific register block

```
component pss_top {  
    transparent_addr_space<> aspace;  
    // ...  
    dma_c                                dut;  
    dma_regs_c                           regs;  
  
    exec init_down {  
        dut.reg = regs;  
    }  
}
```

Connecting PSS to a SystemVerilog Testbench

- Programming sequences interact with IPs via memory and memory-mapped registers
- PSS defines a standard set of read/write routines for accessing memory
 - May be called directly by user-defined test realization
 - Called indirectly when user-defined test realization reads/writes registers
- Implement these read/write functions in terms of target environment
- Direct to BFM
 - Implement read/write methods in terms of your BFM API
- Connect to UVM Reg at address level
 - Implement read/write methods by calling UVM register-model API



PSS at IP-Block Level: Summary

■ Captured

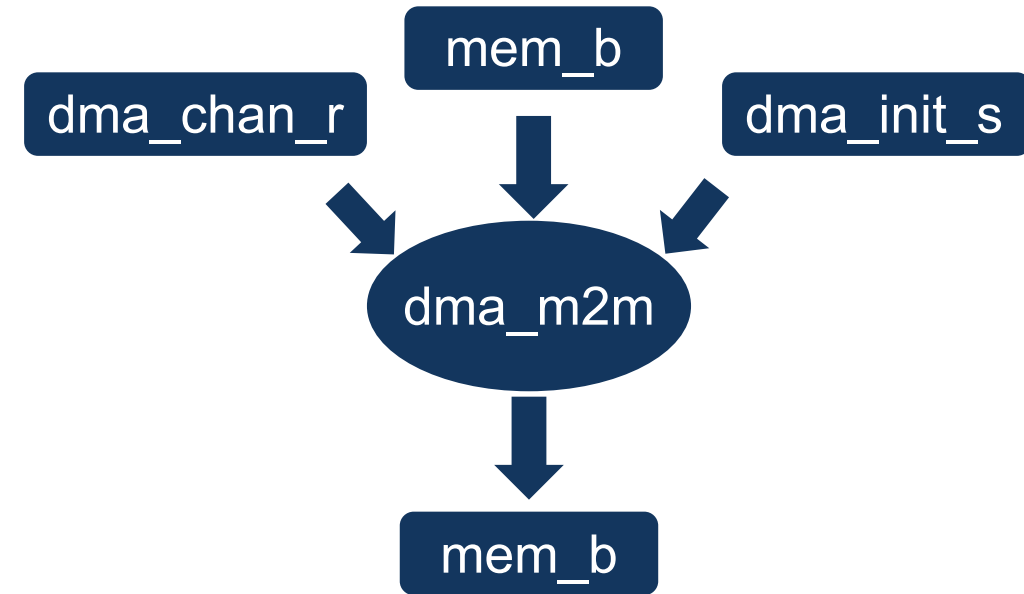
- Test content to Initialize our IPs
- Test content to exercise key behaviors
- Register-access layer to interact with IP registers
- Rules that document our actions' requirements

■ Behaviors capture *requirements* for their execution

- IP must be initialization before use
- Resources required by each behavior
- Memory required by each behavior

■ Requirements Capture+Test Realization = Portability

- Can automatically detect missing requirements (eg missing initialization)
- PSS processing tools can *infer* an action to satisfy the requirement
- Requirements provide automatic documentation





Sub-system and SoC– level testing with PSS

PSS for SoC / Sub-system: Goals and Requirements

- **Focus on integration aspects that are often custom in SoC designs**
 - HW/FW logic that is *prone to bugs*
 - Not verified in lower-level environments
 - Some aspects of desired behavior not explicitly covered in formal specs
 - Subject to “soft” issues, such as overall power consumption and performance, not just clear-cut functional bugs
 - Capture and drive *System-Level* Functional Coverage metrics
- **Typical examples of PSS test content at SoC and Sub-system levels:**
 - SoC integration: coalesce unit tests into cross-IP flows to exercise data paths and system concurrency
 - Can be instrumented for performance measurements
 - Power management and chip bring-up: exercise IP power-cycle sequences and SoC boot flows
 - Generate directed-random sequences of power state transitions on cores/clusters and IPs/subsystems
 - Cross power-related flows with functional “traffic” tests: archetype of system use-cases
 - Additional SoC-level aspects: exercise interrupt controllers, chip frequency switching ...

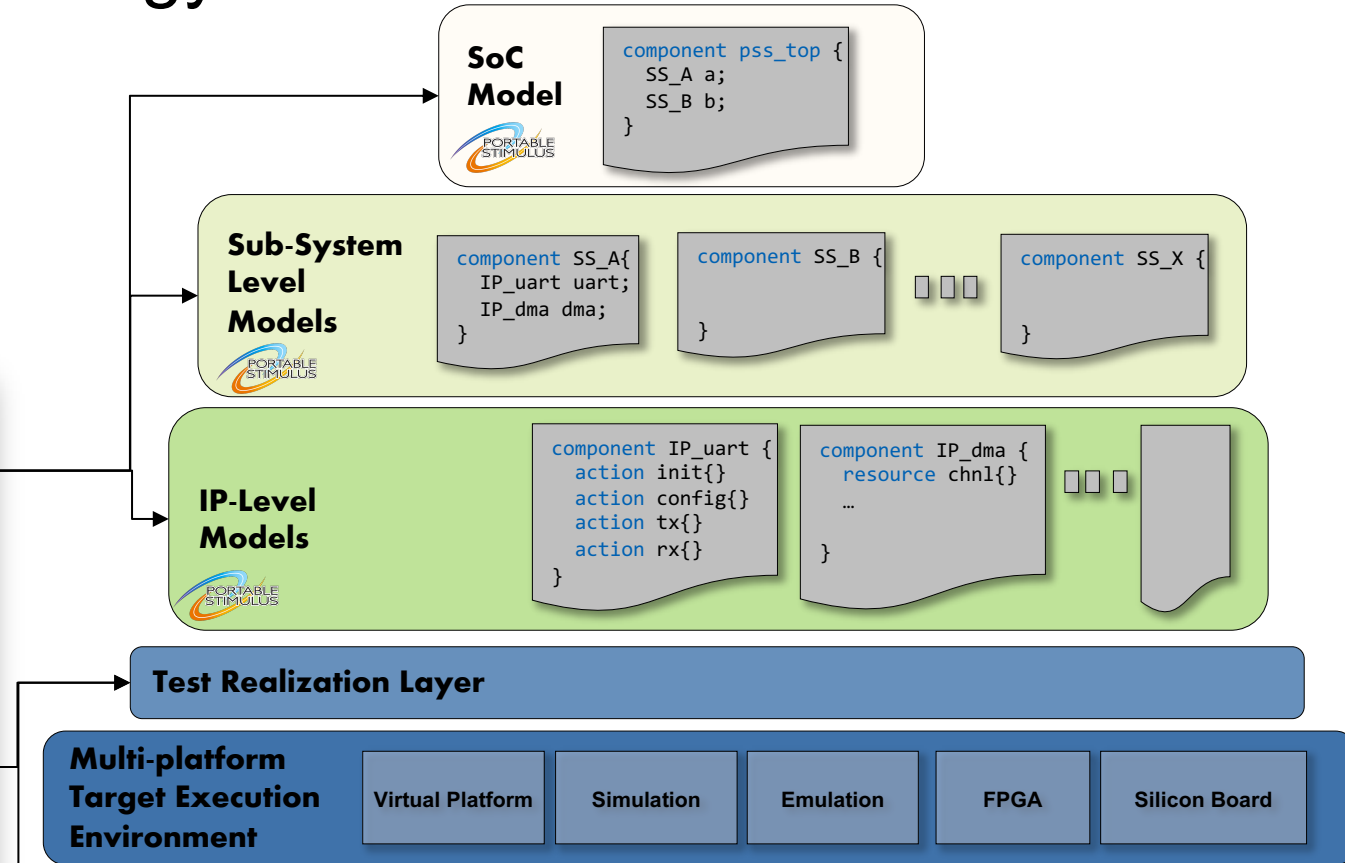
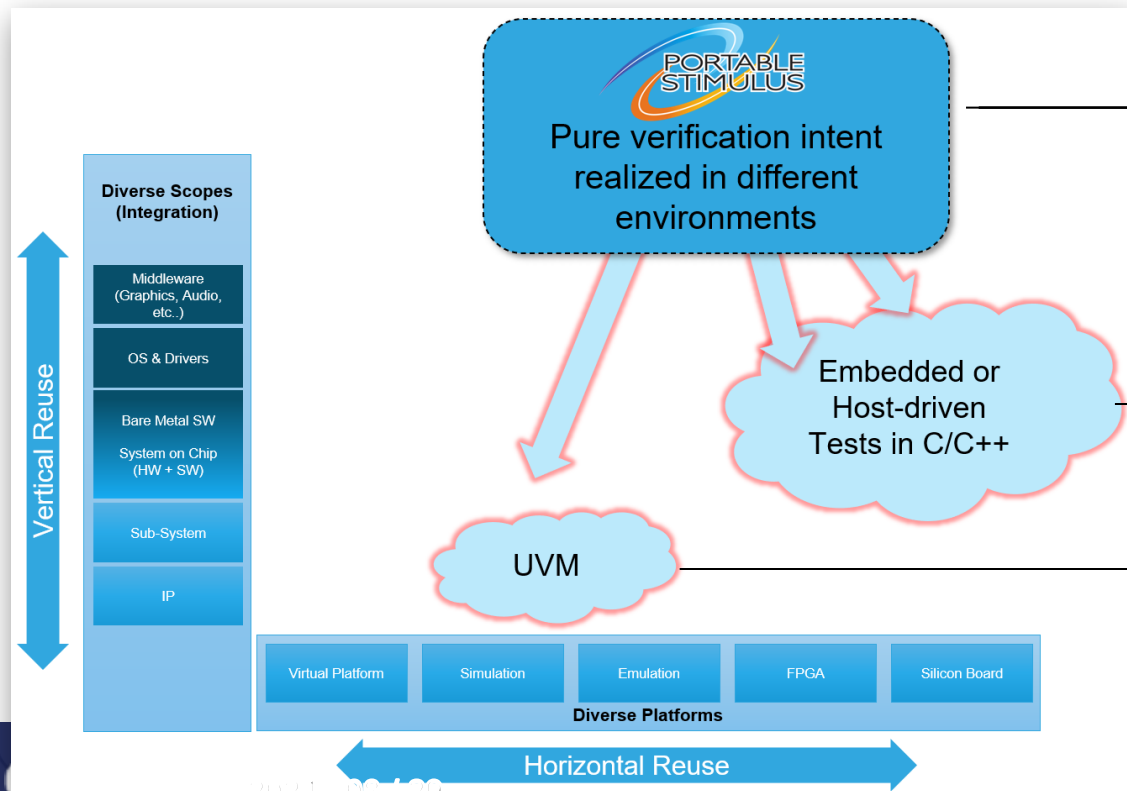
Test Content for SoC / Sub-system: Challenges

- **Facilitate portability across diverse execution platforms**
 - Simulation: coreless with BFMs or processor-driven
 - Fast platforms (emulation): coreless with transactors/AVIPs or processor driven
 - Post-Si: Silicon board, ATE testers - processor driven
- **Quickly initialize required IPs**
- **Generate complex and valid cross-IP traffic patterns**
 - Parallel traffic avoiding resource conflicts
 - Memory allocation management
- **Accommodate changes in register memory maps**
- **Prove coverage of key concurrent behaviors**
- **Unify scenario space model across all testing environments**
 - Reuse abstract test content on transition from register to driver-based testing

Assembling PSS View of a Modern SoC Design: From Vision to Deployable Methodology and Production Use

Typical PSS environment:

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



PSS is agile - any modeling approach is viable!
Top-to-Bottom, Bottom-to-Top or "somewhere in between"
Next slides describe roles and interaction of these layers

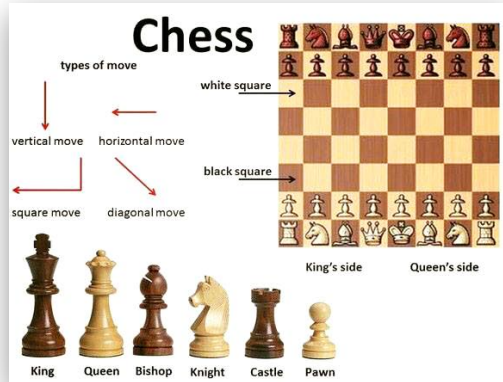
PSS Modeling of SoC - Top to Bottom approach

■ PSS models formally span Test Spaces

- Rules of the game
 - Participating entities, actors and their properties
 - Behaviors, their properties and dependencies

■ PSS activities traverse Test Spaces

- Specific Plays within the game
 - Interesting use-cases, per Test Plan
 - Naturally map onto System-Level Coverage Goals



PSS Modeling mindset

- ✓ Focus on formally spanning test space of SoC
- ✓ Use PSS activities to describe Use-Cases per test-plan
- ✓ Capture key, imperative testing aspects as PSS coverage goals

SoC – Level PSS Use-Cases



Real-world scenarios

- system traffic concurrency
- performance
- power
- cache coherency

SoC - Level PSS model



- #### Structural entities
- Subsystem models
 - IP models

- #### Scenario building blocks and rules
- Power-up and IP initialization rules
 - Basic IP scenarios – initialization, configuration, traffic
 - Cross-IP traffic scenarios



IP and SubSystem Models

Test Realization Layer

Multi-platform Target Execution Environment

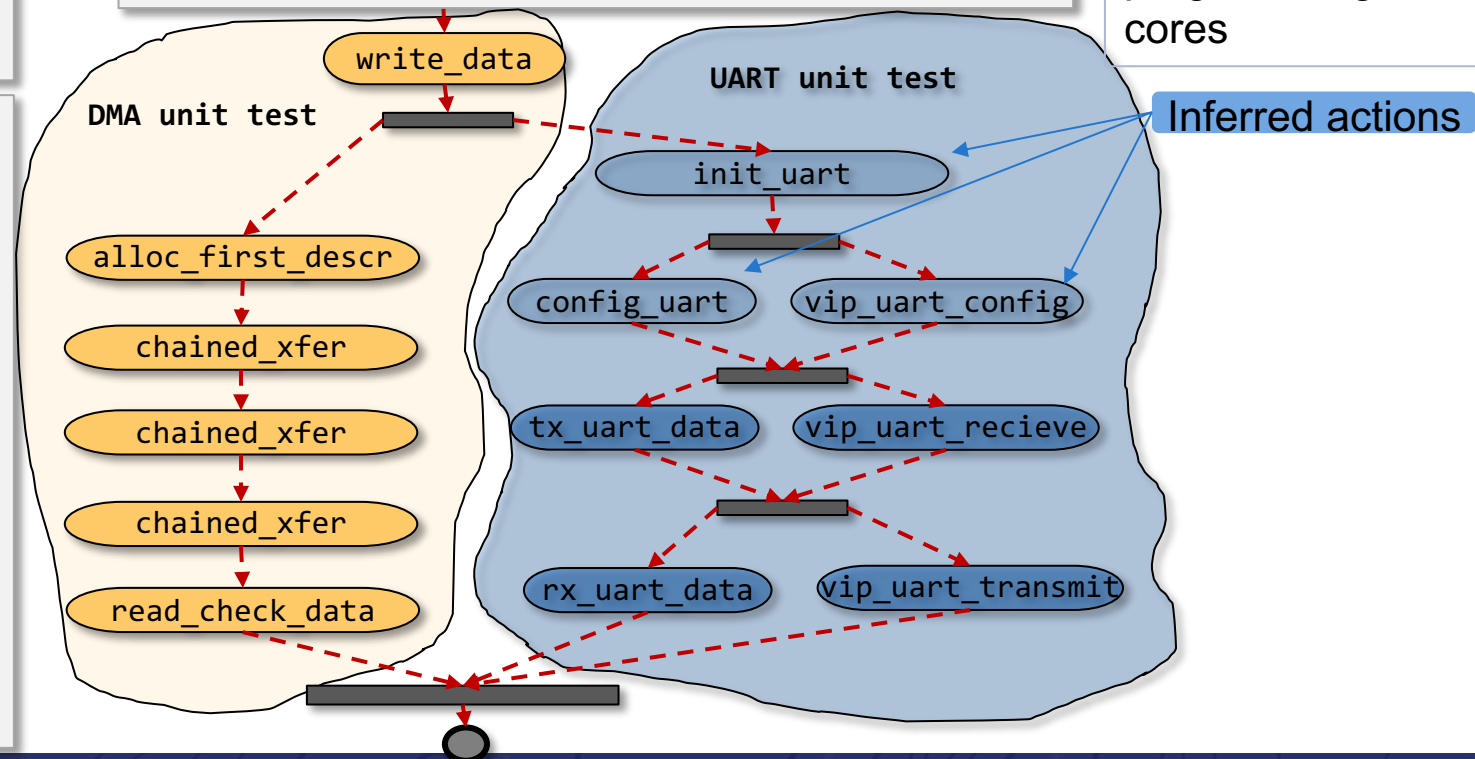
Modeling cross-IP flows at SoC Level

```
// DMA IP-level unit test
action single_dma_xfer_descr {
  activity {
    sequence {
      do psm_memory_ops_c::write_data ;
      do dma_c::descr_xfer;
      do psm_memory_ops_c::read_check_data;
    }
  }
}
```

```
// UART IP-level unit test
action uart_tx_rx {
  activity {
    sequence {
      parallel {
        do uart_c::tx_uart_data;
        do uart_vip_c::vip_uart_receive;
      }
      parallel {
        do uart_c::rx_uart_data;
        do uart_vip_c::vip_uart_transmit;
      }
    }
  }
}
```

```
// SoC - level Use-Case
action test_uart_with_dma {
  activity {
    schedule {
      do dma_c::single_dma_xfer_descr;
      do dma_c::uart_tx_rx;
    }
  }
}
```

Solve for legal, random scheduling of sub-activities. Perform resource and memory allocation, assign programming cores



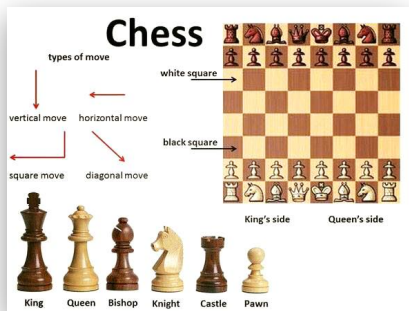
PSS Modeling of SoC – Let There be IP !

SoC – Level Model



- Scenario building blocks and rules
- Power-up and IP initialization rules
 - Basic IP scenarios – initialization, configuration, traffic
 - Cross-IP traffic scenarios

- Real-world scenarios
- system traffic concurrency
 - performance
 - power
 - cache coherency



Moving on to development of IP-level models
PSS modeling mindset remains the same!

- ✓ Focus on formally spanning test spaces of each IP
- ✓ Use PSS activities to describe IP Use-Cases per test-plan
- ✓ Capture key, imperative testing aspects as PSS coverage goals



IP-Level PSS models



- Specify IP connectivity, resources, SW contracts
- Object pools and action bindings
- Capture IP Configuration aspects
- Describe unit-level use cases and behaviors
- Init, config, traffic ...

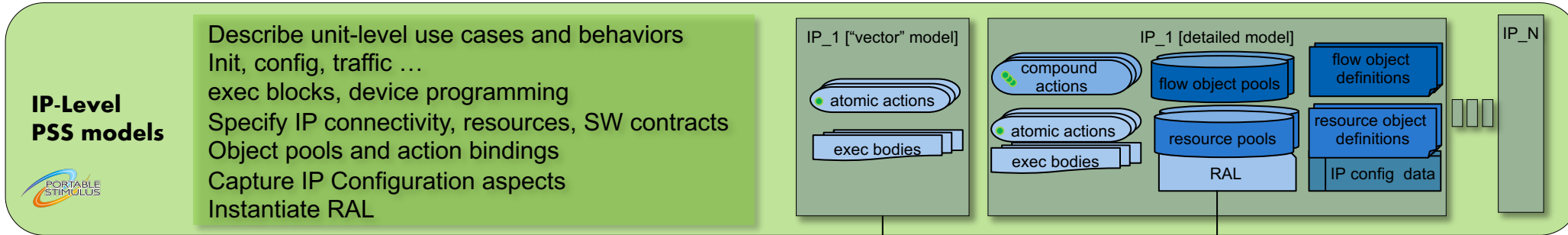
- Test Realization Layer Interface
- exec blocks
 - device programming sequences
 - instantiate RAL

Test Realization Layer

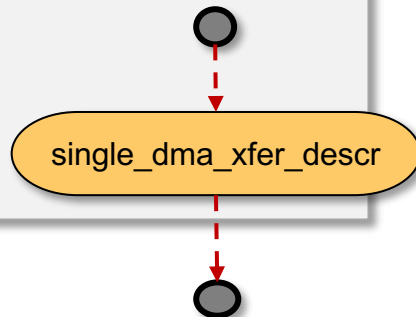
Multi-platform Target Execution Environment

PSS Modeling of SoC – Devil is in the Detail ...Really?

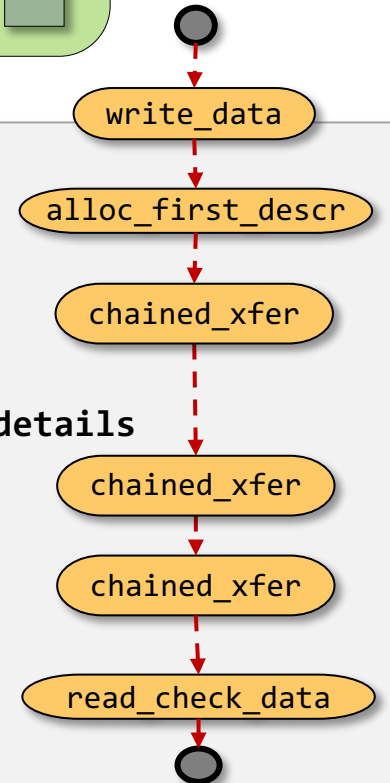
IP models level may vary in level of detail: **coarse, abstract** ↔ **fine-grained, detailed**



```
component dma_c {  
  // Import "vector" test functionality  
  target function void single_dma_xfer_test();  
  import function single_dma_xfer_test;  
  // DMA IP-level unit test:  
  action single_dma_xfer_descr {  
    // Atomic action-simply invoke a test vector  
    exec body {  
      single_dma_xfer_test();  
    }  
  }  
}
```

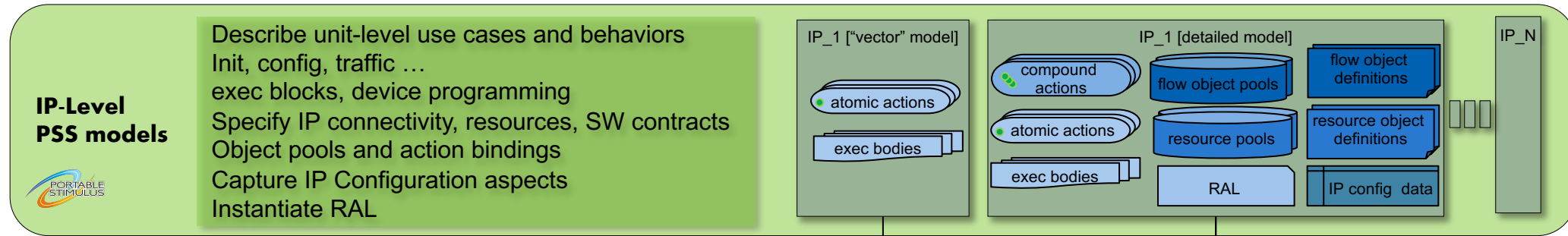


```
component dma_c {  
  resource chnl_r {}  
  pool [NUM_CHANNELS] chan_r;  
  action descr_xfer { lock chnl_r chnl; ...}  
  // DMA IP-level unit test  
  action single_dma_xfer_descr {  
    // Compound activity scope refines use case details  
    activity {  
      sequence {  
        do psm_memory_ops_c::write_data;  
        do dma_c::descr_xfer;  
        do psm_memory_ops_c::read_check_data;  
      }  
    }  
  }  
}
```



Methodology - Who Owns PSS IP-level Models

IP ownership methodology choice depends on project phase and degree of PSS technology adoption across different teams



initial adoption of PSS

SoC team uses IP test vectors, provided by IP teams

- “test vectors” - existing tests in C or SV
 - supplied as precompiled binaries or source C code
- Pros:
- Leverage basic PSS for SoC-level scenario composition
 - Cheap and quick to deploy (nobody owns IP models)
- Cons:
- Limited controllability (vector tests are monolithic)
 - Hard to inter-operate with other unit tests

Widening adoption of PSS

SoC team owns IP level models

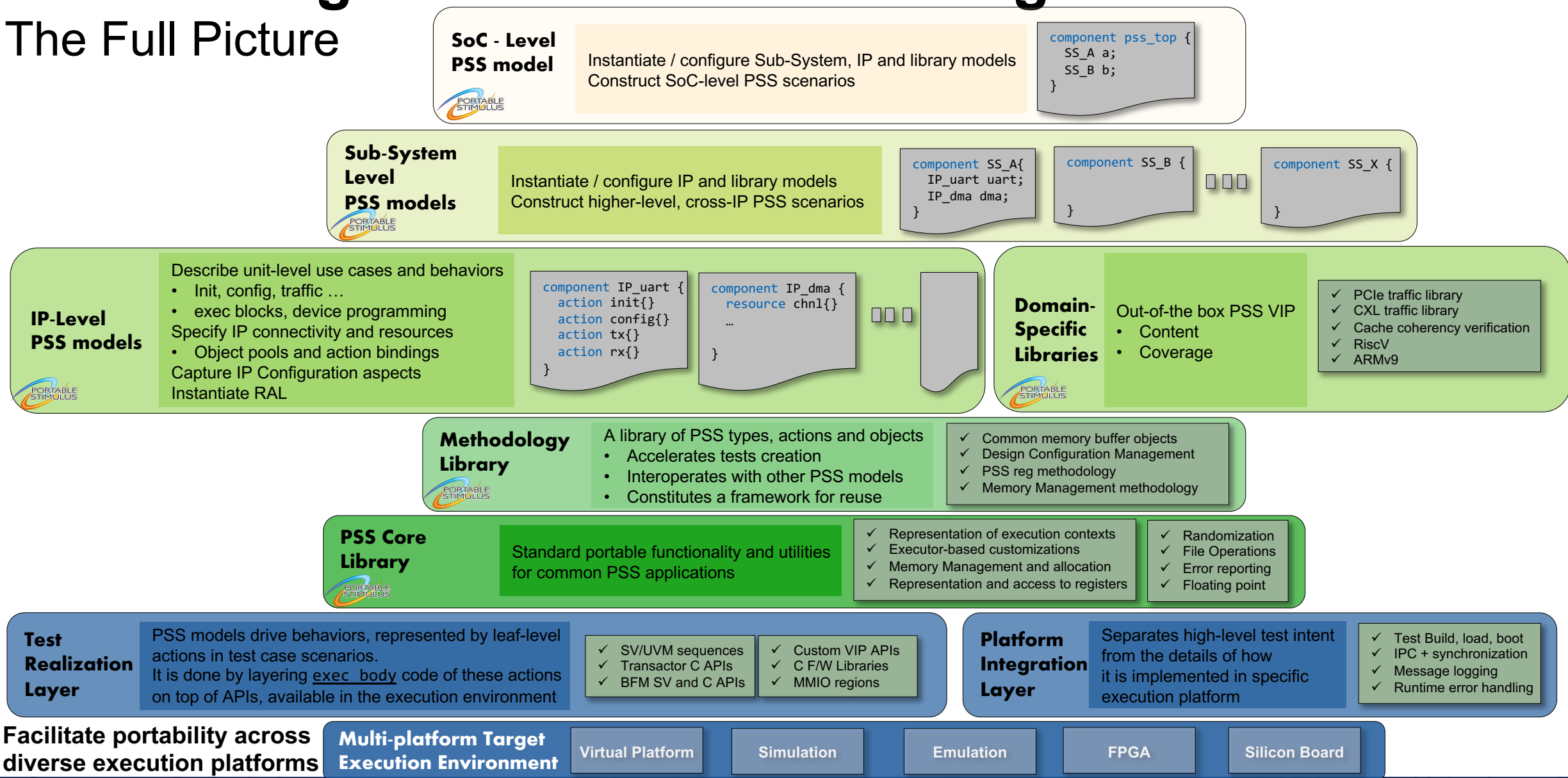
- Pros:
- Leverage full power of PSS - better controllability and inter-operability with other unit tests
- Cons:
- No benefit for IP teams
 - Higher upfront investment for SoC team

Company-wide adoption of PSS

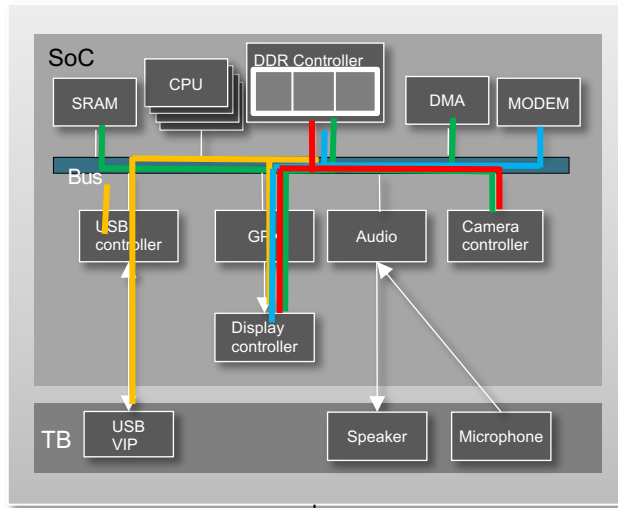
IP teams own IP level models, reused by SoC team

- Pros:
- Leverage full power of PSS
 - Benefits IP teams as well
- Cons:
- Upfront investment for IP teams

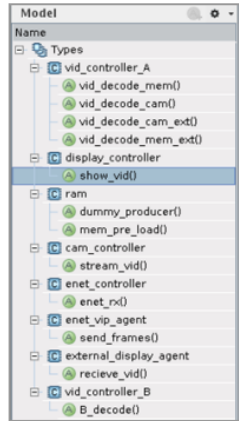
Assembling PSS View of SoC Design - The Full Picture



Modeling cross-IP flows with PSS activity statements



PSS Model



Abstract PSS scenarios may be specified partially

```
// Video SoC-level test
action video_scenario {
  activity {
    do camera_c::capture;
    do dma_c::transfer;
    do gpx_c::decode_to_display;
    do display_c::show;
  }
}
```

```
// Audio SoC-level test
action audio_scenario {
  activity {
    repeat (3) {
      do modem_c::receive;
      do audio_c::play;
    }
  }
}
```

Focus on certain memory areas

Randomly distribute actions on available cores

Find legal configurations, consistent with all use cases

```
action mixed_scenario {
  activity {
    parallel {
      do video_scenario;
      do audio_scenario;
    }
  }
}
```

Behavioral Activity Statements

Correct by construction:
PSS solvers ensure no over-use of available resources, insert sync points if/as needed to achieve a legal scenario



Generate complex and valid cross-IP traffic patterns

PSS Coverage: Value-add, Differentiation and ROI

Value-add

- Construction and analysis of **system-level** functional coverage metrics
 - Portable
 - Abstract
- Predictability
 - Regression suite optimization, faster coverage closure with gen-time coverage prediction

Differentiation

- Enabler of innovative verification methodologies and flows

Cost

- Coverage methodology is consistent with PSS scenario modeling and test generation mindset
- Trainable, deployable

Proven Impact and Differentiation
Low Deployment Cost
= High ROI =

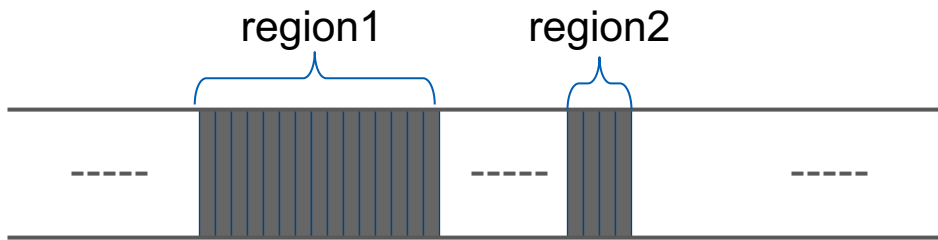
PSS Coverage: Current Capabilities and Applications

- **Abstract and high-level, like PSS stimulus itself**
 - Required for applications in system level, use-case based and software-driven validation
- **Portability: critical enabler for applications on fast platforms with low observability**
 - ATE, Emulation, Silicon Boards, bare metal environments
- **Enables tools to predict coverage at generation time**
 - Possible because PSS scenarios are declarative, can be solved upfront
 - Highly differentiated in comparison to UVM and other procedurally-driven environments
 - Enables flows aimed at generation of exhaustive coverage regression suites
- **Easy to define functional coverage spaces over PSS scenario attribute values**
 - Structurally, same as SV coverage – sets of combinations expressed in terms of cover points, bins and crosses
 - Interoperable with eco-system: other coverage engines .e.g Formal, SV and test plan tracking databases
 - Low adoption barrier and deployment cost
 - Can define coverage goals that span across multiple actions within a scenario
 - Enables specification, collection and tracking of system-level behavioral coverage goals [new in PSS 3.0]

Memory Allocation Consistency in PSS

```
action my_op {  
  rand addr_claim_s<> claim;  
  constraint claim.size == 20; }  
}
```

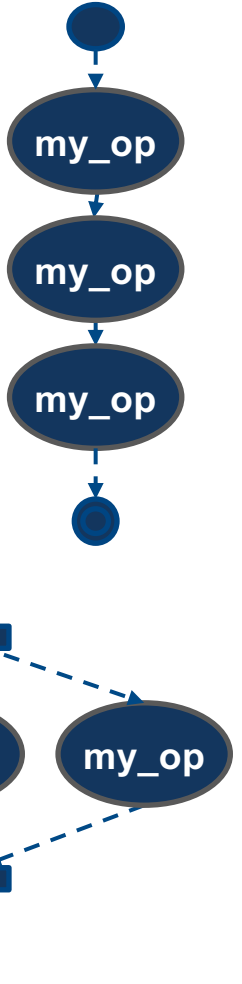
```
component pss_top {  
  action my_op {  
    rand addr_claim_s<> claim;  
    constraint claim.size == 20;  
  }  
  contiguous_addr_space_c<> mem;  
  exec init {  
    addr_region_s<> region1, region2;  
    region1.size = 50;  
    mem.add_region(region1);  
    region2.size = 10;  
    mem.add_region(region2);  
  } }  
}
```



```
action test1 {  
  activity {  
    repeat (3) {  
      do my_op;  
    }  
  }  
}  
  
action test2 {  
  activity {  
    parallel {  
      replicate (3) {  
        do my_op;  
      }  
    }  
  }  
}
```

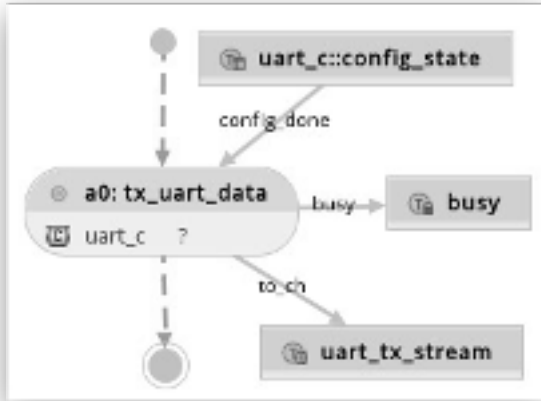
OK – allocations can be recycled across sequential claims

Allocation error! – cannot satisfy concurrent claims



✓ Memory allocation management

Methodology: Unify Scenario Space Model Across Environments



```
// UART TX action
action tx_uart_data: uart_base {
  output uart_tx_stream to_ch ;
  input config_state config_done;
  constraint config_done.done == true;
  lock uart_tx_status_r busy ;
  rand bit[8] data;
  constraint to_ch.data == data;
}
```

PSS is aspect oriented. **extend** high-level, abstract representation of behaviors (actions) to provide **different** implementations of their `exec body` blocks, capturing low-level device programming logic

```
extend action tx_uart_data {
  exec body {
    uart_ctrl_ua_tfifo_reg_s tfifo;
    tfifo.data[7:0] = data;
    comp.regs.ua_tfifo.write(tfifo);
    while comp.regs.ua_csr.read().empty == 0 {
      message(NONE, "Checking Transmit done");
      yield;
    }
  }
}
```

Write and read memory-mapped H/W registers

```
extend action tx_uart_data {
  exec body C = ""
    struct UartConsoleDriver *uart_console =
      &(UartConsole[0]);
    struct UartDriver *uart_driver =
      &(uart_console->driver);
    Uart_WriteTxData(uart_driver, {{data}});
    while (1) {
      if (Uart_TxFifoEmpty(uart_driver))
        break;
    }
  ""
}
```

Reuse abstract PSS content on transition from register to driver-based testing

Call C S/W Driver APIs

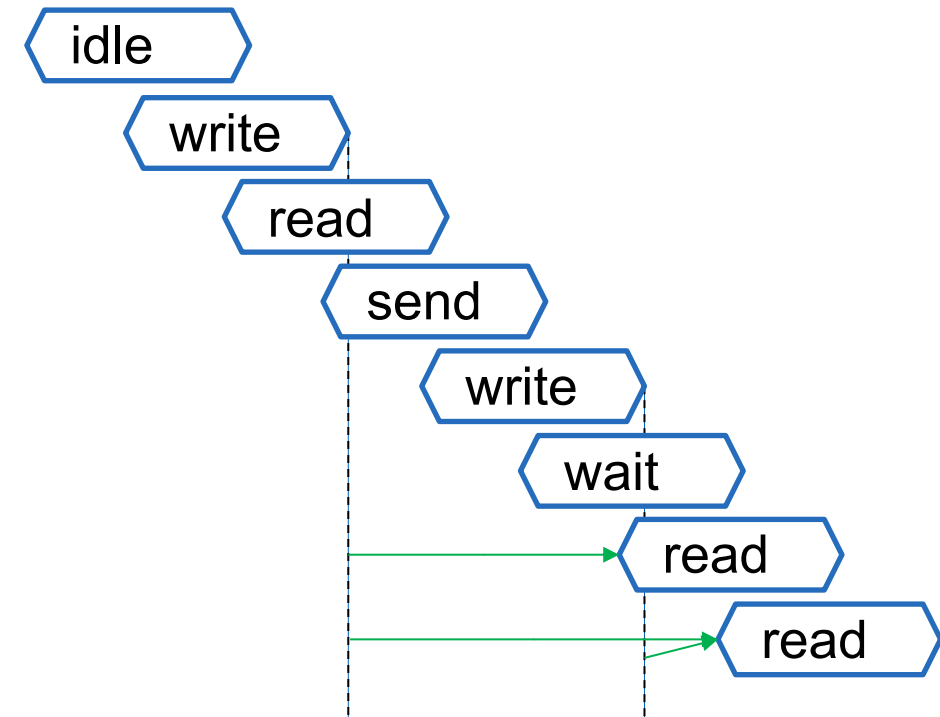


PSS 3.0 & Wrapping Up

What's Coming in 3.0

Introducing Scenario-Level Behavioral Coverage

- Given a stream of action executions, find out whether a given temporal scenario (query) occurs in this stream
- The *cover* statement specifies the interesting scenario
- A *monitor* encapsulates behaviors to be covered
 - A monitor may be implicit (in a cover statement) or explicit
- The answer is yes or no
 - Yes, if the top-level monitor has at least one match,
 - No, otherwise



WR: cover { do write; do read }

New in PSS 3.0: Behavioral Coverage

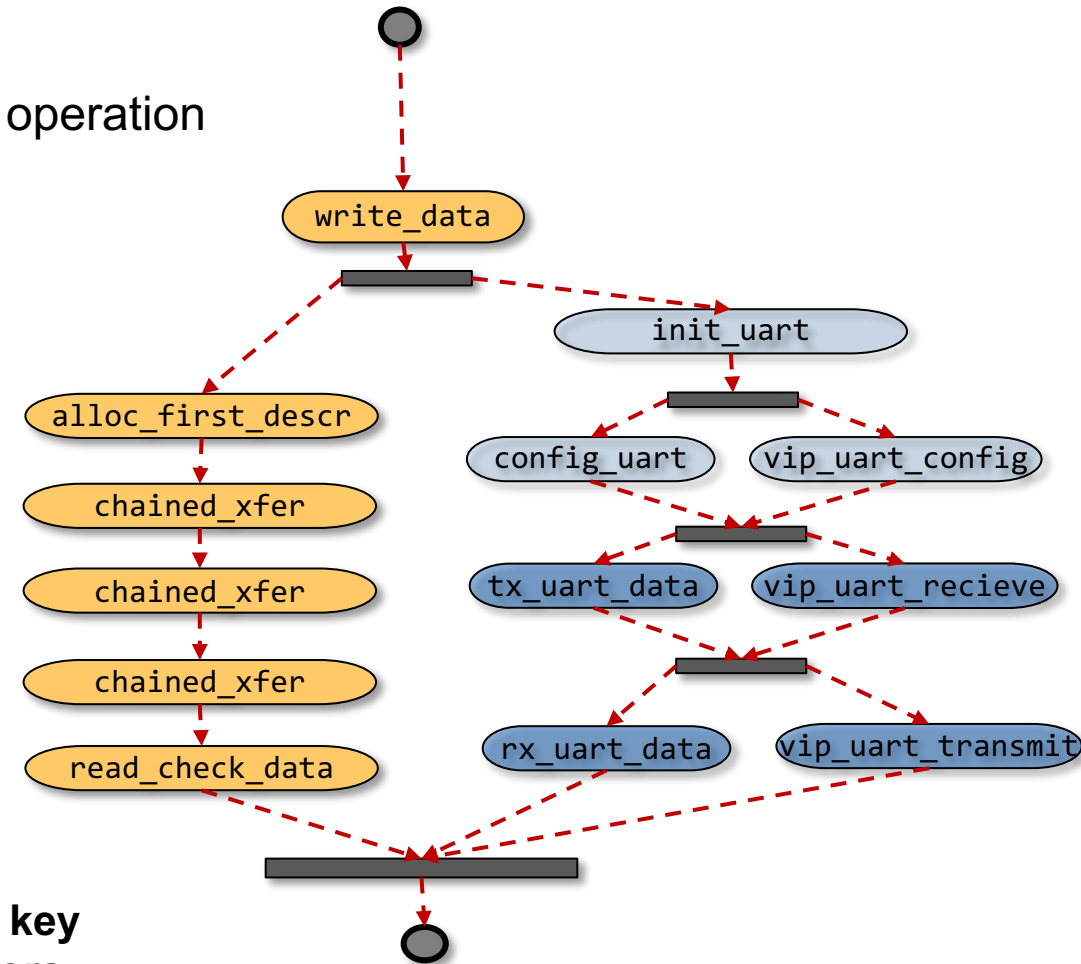
■ System-Level coverage goal:

- observe overlapping execution of DMA transfer and UART TX operation
- Cross-cover different cores

```
// Behavioral Coverage Monitor
c: cover {
  activity {
    overlap {
      tx: do uar_c::tx_uart_data;
      dma: do dma_c::chained_xfer;
    }
  }
  covergroup {
    tx_proc: coverpoint tx.core.tag;
    dma_proc: coverpoint xfer.core.tag;
    tpXdp: cross tx_proc, dma_proc;
  } cg;
}
```



Prove coverage of key concurrent behaviors



PSS 3.0 - Public Review

- Behavioral Coverage, which allows the user to identify a set of actions and data combinations that need to be observed to exercise key functionality.
- String methods, including size() and find(), and the sub-string operator to extract a sub-string from a given string (7.6)
- Support for collections of reference types (7.10)
- Platform qualifiers on function prototype declarations (22.2)
- Support for comments in target-template blocks (22.5)
- Support for yielding control with cooperative multitasking (22.7.14)
- Added an address space group to allow multiple address spaces to share common storage elements (24.9)
- Defined mapping between PSS lists and SystemVerilog Queues (D.5.5)
- Added Annex F to specify the formal semantics of behavioral coverage

Summary: PSS Advantages for SoC Verification Engineers

Productivity



Improves test generation throughput.

Exchange test intent model across multiple teams

Target multiple cores and HVL testbenches

Abstraction



High-level description of test intent in a concise model

Strong semantics to capture memory, resource dependencies

Easy to reason, communicate and analyze test scenarios

Reuse



Leverage tests across other platforms, avoid duplication and enable reuse

Portability of tests across projects by clean separation of configuration data from behavioral model

Quality

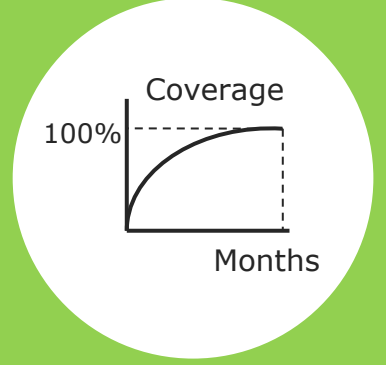


Correct by construction test scenarios with concurrency, synchronization, deep dependencies, resource management

Action inference automates completion of partially defined scenarios into legal, concrete test cases

Abstract debug of system test failures

Coverage Closure



Faster Coverage closure with upfront coverage analysis

Efficient regression planning: Generate and execute test that contribute to coverage

Collect and analyze System-level coverage from multiple execution platforms

Improve and optimize schedule, quality, machine and human resources!!

Questions ?