# PSS Comes of Age:
# Runtime Behavioral Coverage, Methodology and More

Accellera PSS Working Group

accellera SYSTEMS INITIATIVE™

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

# PSS Runtime Behavioral Coverage – What is it?

- In PSS 2.0 we introduced Data Coverage:
  - Used to report data coverage
  - Like System Verilog cover-groups
  - Primary Application is at solve time
- In PSS 3.0 we introduced runtime behavioral coverage:
  - Used to report coverage of behaviors of actions
  - Primary application is for runtime coverage with behavior and data
  - Evaluated over a runtime trace of events
  - LRM formally specifies the expected coverage results given a coverage PSS model and a trace of events
  - Usage of PSS Monitors to model activities required to observe to meet coverage goals
  - PSS Actions used to model activities for generating action traversal behaviors
  - PSS Monitors used to model activities for detecting action traversal behaviors
  - PSS Monitors have the look and feel of PSS Action. Making the adoption easier

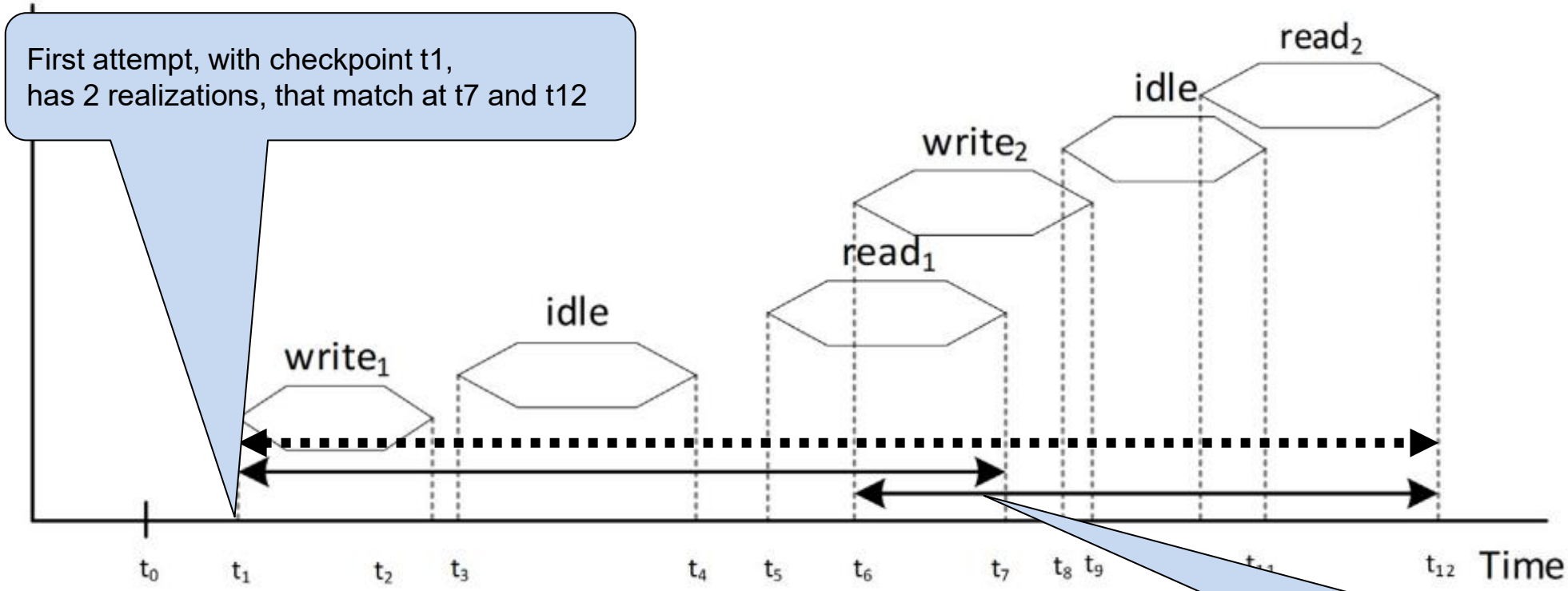
# PSS Runtime Behavioral Coverage – Example

```
monitor m1 { write w; read r; activity { w; r; } }
c1: cover m1;
```

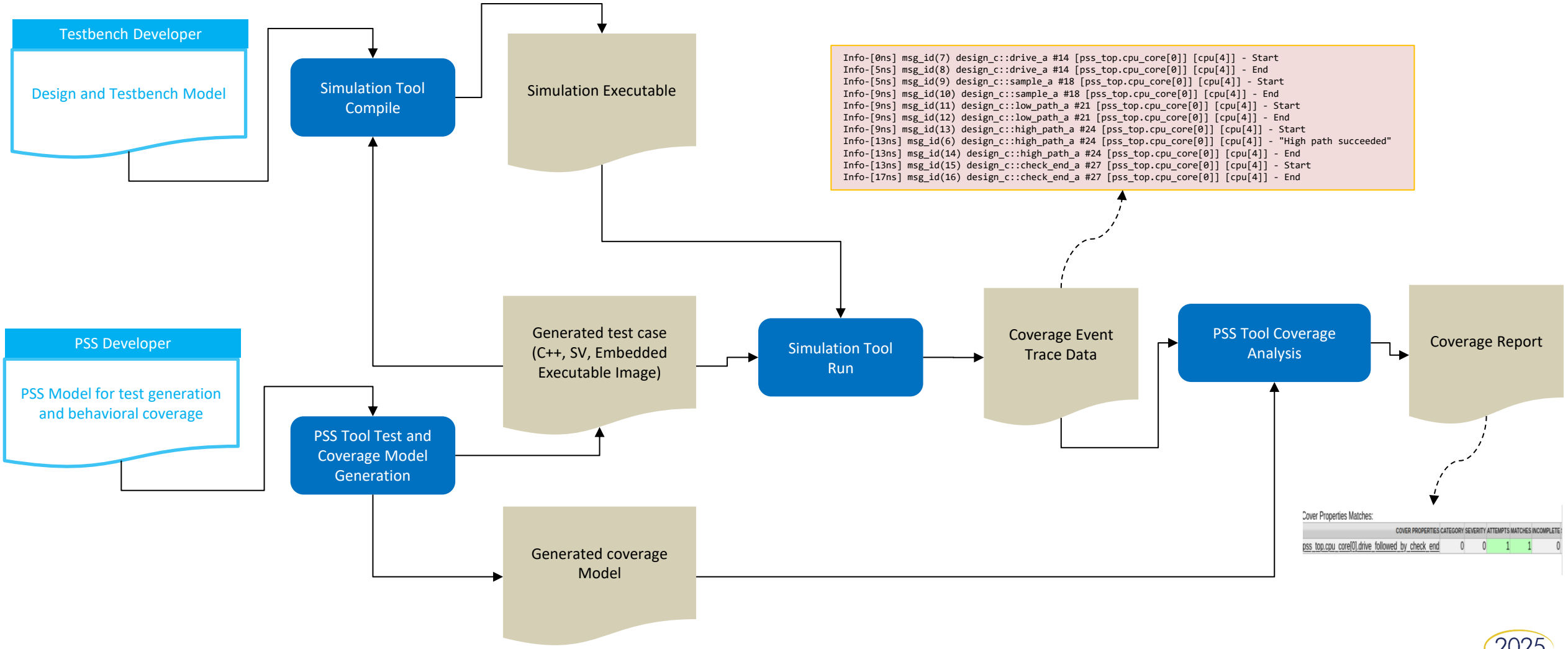
A *scenario checkpoint* is a time instant relative to which the scenario is observed.

An *attempt* is a scenario along with its checkpoint.

An *attempt scenario realization* is a set of action executions traversed by this attempt along with the mapping of action handles into specific action executions.



# Runtime Behavioral Coverage Simulation Flow Example



# PSS Runtime Behavioral Coverage – Motivation and Benefits

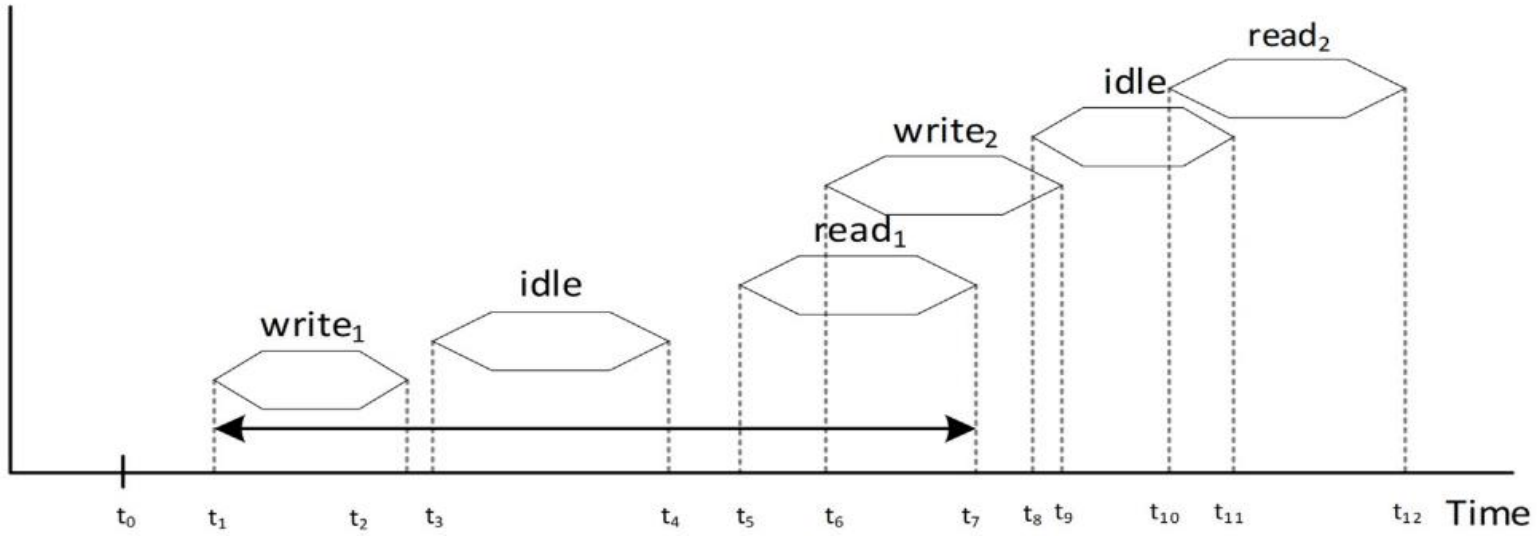
- Model monitors to detect behavioral patterns of action traversals
- Enables tools to report coverage of desired behavioral patterns
- Enables sampling behavioral patterns together with data combinations
- Desired overlapping behaviors can only be confirmed at runtime
- Portable across simulation, emulation, silicon, field … All that's needed is a trace for the required types
- Enables post processing flow, to sample coverage
- Enables sampling data updated at test runtime
- Modular modeling using monitors hierarchically, just like actions
- Usage of constraints to refine pattern detection based on data
- Usage of action like syntax to minimize learning new language (Actions are used for behavior generation. Monitors are used for behavior observation and detection)
- Provides analysis information for incomplete runs

# Monitor “sequence” activity operator

```
monitor m1 {
  activity {
     sequence {
         do write;
         do read;
     }
  }
}
```

- For a *sequence* to match
  - The specified actions must be traversed in order
  - They may not overlap in time
- Other concurrent action traversals do not invalidate the match
  - Ex: ‘idle’ does not invalidate the sequence write, read

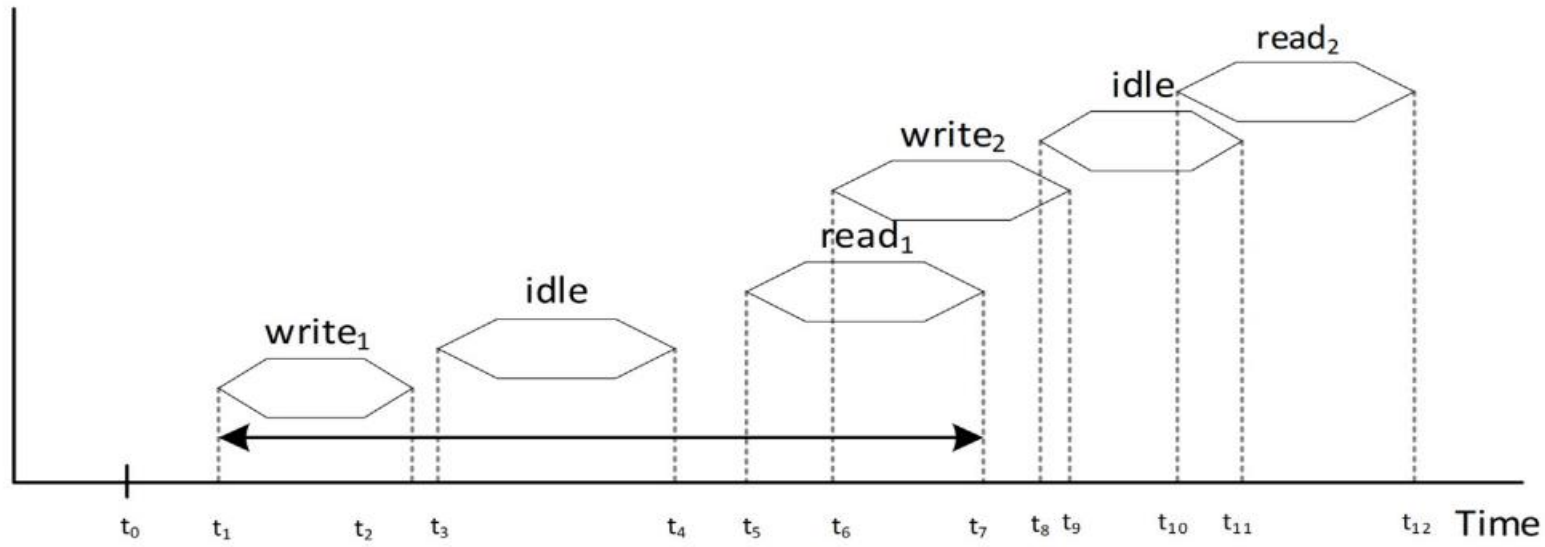
1



# Monitor “concat” activity operator

The concatenation scenario defines an immediate consecutive matching of its sub-scenarios; the checkpoint of the next sub-scenario is the matching point of the previous one.

```
c1: cover { activity { concat { do write; do read; } } }
c2: cover { activity { sequence { do write; do read; } } }
```



# Monitor inline constraints

```
c5: cover { activity { concat { do start; do write with core == 0; do read; } } }
c6: cover { activity { sequence { do start; do write with core == 0; do read; } } }
```

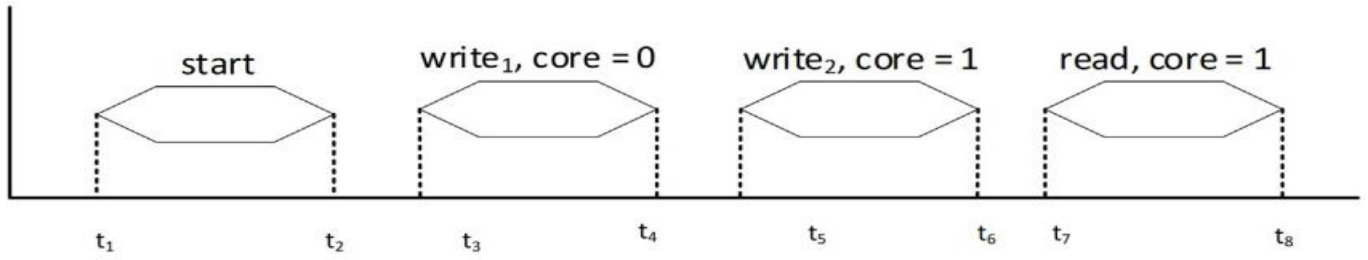



**Figure 31—First alternative.** Example 195

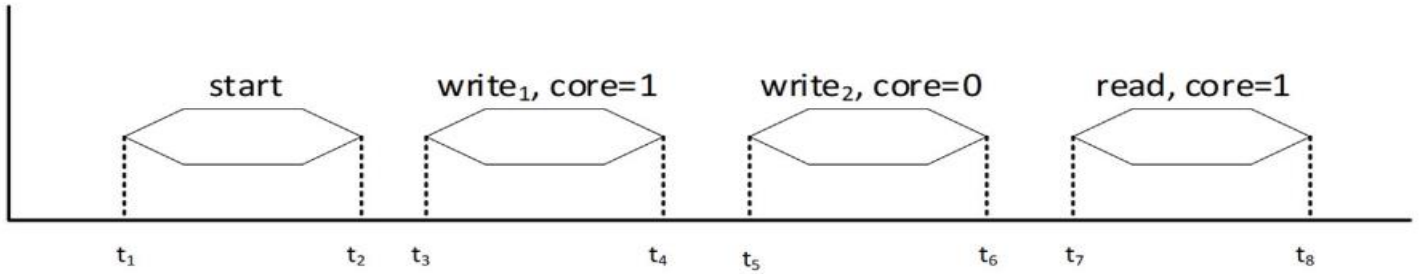



**Figure 32—Second alternative.** Example 195

- The top-level scenarios of c5 and c6 have the same realization for each trace: {start, write1, read} for the trace in Figure 31 and {start, write2, read} for the trace in Figure 32
- The inline constraint in the concat scenario in cover statement c5 will match the first traversal of write that satisfies the constraint. In Figure 32, starting at t2, the match point of the start traversal, the subscenario “do write with core == 0” has the realization {write2}

# Monitor “schedule” operator

```
action read {}
action write {}
action send {}
action receive {}
monitor m1 { activity { schedule { sequence { do read; do write; }; sequence { do write; do send; } } }
monitor m2 { activity { schedule { sequence { do write; do send; }; sequence { do send; do receive; } } }
```

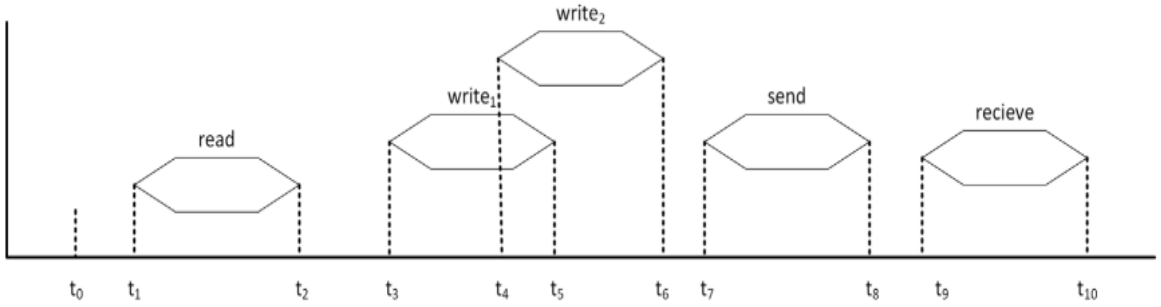



**Figure 44—Scheduling scenario with common actions**

- The scheduling scenario defines execution of its sub-scenarios in any order, if scenario realizations of the member scenarios are not shared
- There may be any overlaps or gaps between its member scenario spans
- In the scheduling scenario, the sub-scenarios are matched from the checkpoint of the scheduling scenario or after it
- The checkpoints of individual sub-scenarios are independent of each other
- The realizations of scenario schedule { s1,…,sn } consist of member-wise realization unions of sub-scenarios s1,..., and sn, provided that these realizations of different scenarios are pairwise disjoint
- For example, if set {a,b} is a realization of s1, set {c} is a realization of s2 and set {d,e,f} is a realization of s3, then set {a,b,c,d,e,f} is a realization of schedule { s1,s2,s3 }; here, a, b, c, d, e, and f are action executions

# Monitor overlap operator

```
action read {}
action write {}
monitor m { activity { overlap { do read; do write; } } }
```

- The overlapping scenario defines overlapping of its member sub-scenarios
- The overlapping scenario meets the same conditions as the scheduling scenario and an additional condition
- The additional condition is that there is a time instant where all its member scenarios are simultaneously active

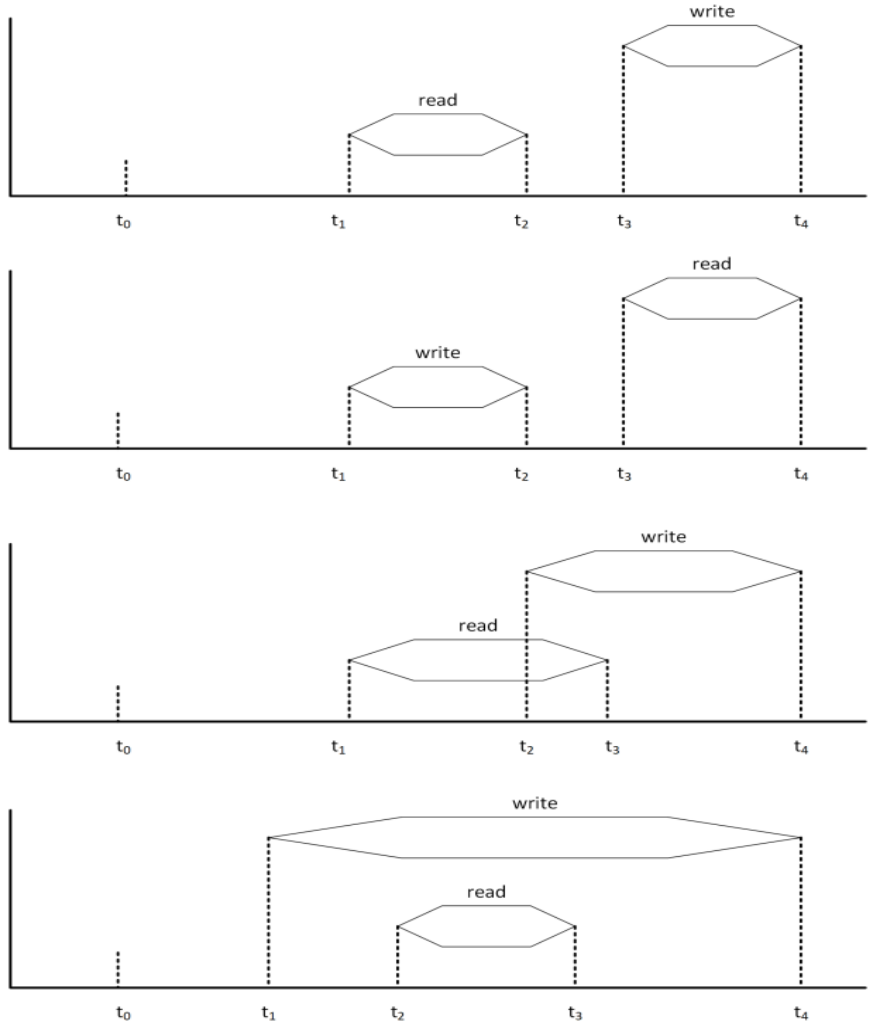
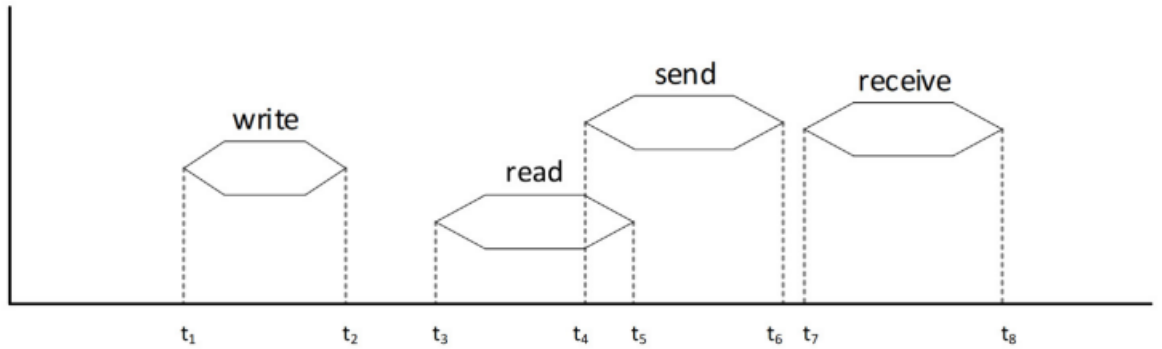



**Figure 39—Overlapping scenario**

# Monitor select operator

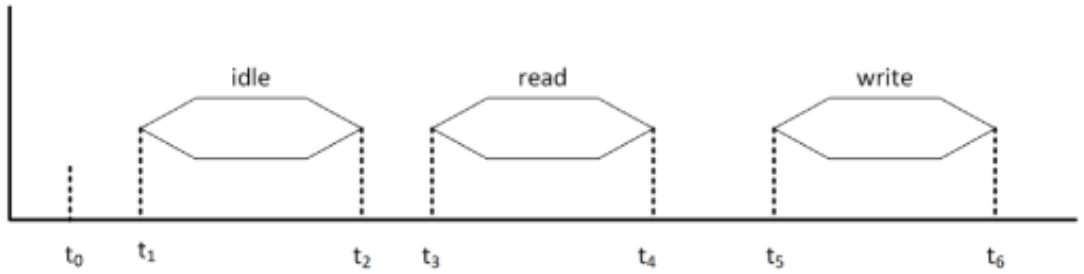
```
action read {}
action write {}
action idle {}
action send {}
action receive {}
c1: cover {
  activity {
    do write;
    select { do read; do send; };
    do receive;
  }
}
c2: cover {
  activity {
    select { do write; do read; };
    select { do send; do receive; };
  }
}
```




- Cover statement c1 has a successful attempt starting at t1 with two realizations: {write, read, receive} and {write, send, receive}
- Cover statement c2 has two successful attempts, one starting at time t1 with two realizations {write, send} and {write, receive}, and the other has starting at time t3 with the realization {read, receive}

# Hierarchical Monitors for composing coverage Models

```
action idle {}
action read {}
action write {}
monitor rw { activity { do read; do write; } }
monitor irw { idle i; read r; write w; activity { i; r; w; } }
monitor irw1 { idle i; activity { i; do rw; } }
monitor irw2 { idle i; rw m; activity { i; m; } }
c1: cover irw;
c2: cover irw1
c3: cover irw2
```




- Cover statement c1, c2 and c3 are equivalent
- Monitors irw, irw1, and irw2 are equivalent
- Consider the monitor matching in the trace for the checkpoint t0
- The monitor irw specifies a sequential scenario and its only realization is {idle, read, write}
- Monitor irw1 defines a sequential scenario whose sub-scenarios are the traversal of action i with the realization {idle} and the traversal of an anonymous monitor of type rw
- The monitor type rw, in its turn, defines a sequential scenario, and its checkpoint is t1 or later so that the realization of monitor irw1 is {idle, read, write}
- The monitor irw2 differs from the monitor irw1 only in that it traverses the monitor of type rw using its handle m, and of course, has the same realization {idle, read, write}

# Covergroups in Monitors

```
enum locked_e { LOCKED, UNLOCKED };
enum write_mode_e { WRITE_BACK, WRITE_THRU };
action read { rand locked_e lock_mode; }
action write { rand write_mode_e write_mode; }
c: cover {
  write w;
  read r;
  activity { w; r; }
  covergroup {
    cpw: coverpoint w.write_mode;
    cpr: coverpoint r.lock_mode;
    wXr: cross cpw, cpr;
  } cg;
}
```

- Covergroups may be defined and instantiated in monitors and cover statements to collect data coverage along the scenario defined by the monitor
- A monitor covergroup is sampled at the first match of the attempts of a cover statement where the monitor is traversed (directly or not)
- The sampling is done according to the action handle mapping associated with a first match scenario realization
- If there are several first match scenario realizations, any realization may be selected for sampling by the implementation

# Summary

- Provided understanding of the flow and motivation for using behavioral runtime coverage, motivations included:
- Provided enough tools to make you LRM behavioral runtime coverage literate
  - Went over text mentioned in LRM.
- Challenged you on whether behavioral runtime coverage is something to be used on your next project
  - Are you convinced Runtime Behavioral Coverage  can help you? Why?

# PSS METHODOLOGY LIBRARY

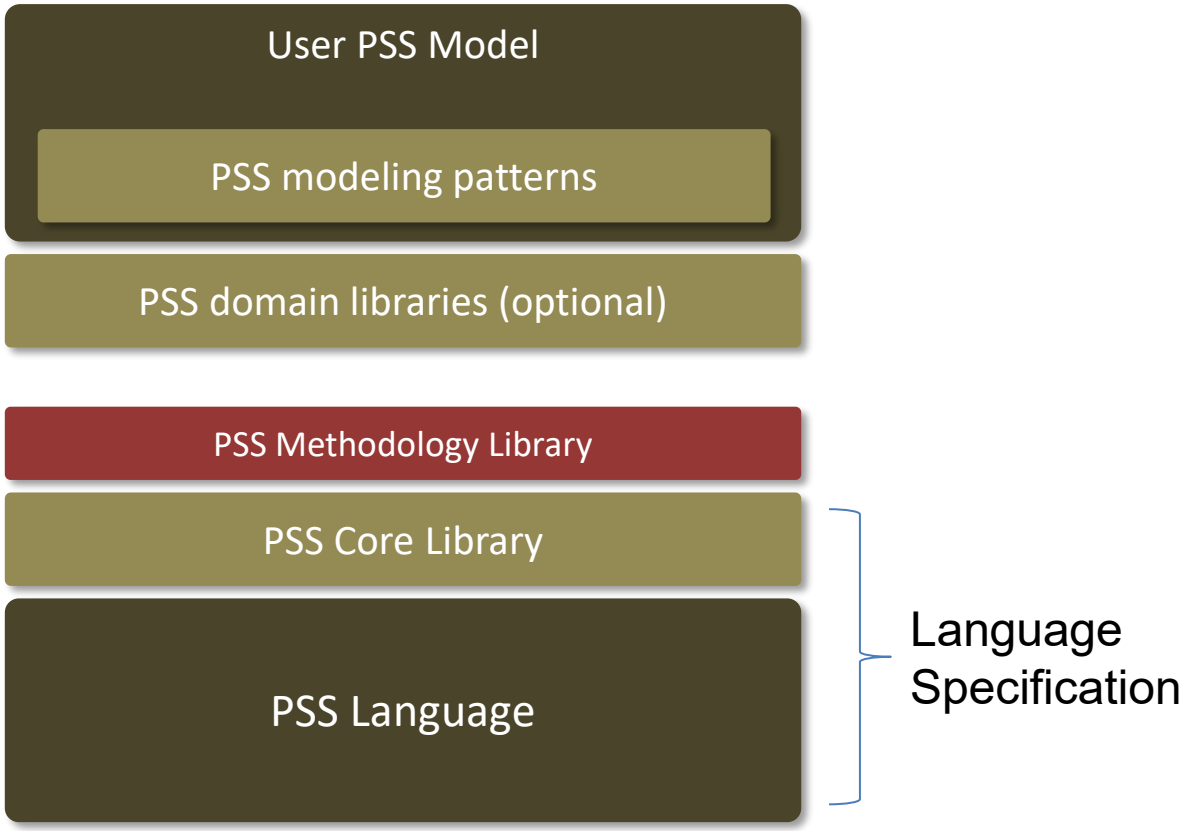
# Agenda

PSS Methodology Library - PSSM

Common test space modeling patterns as motivation for PSSM

# PSS Methodology Library – PSSM*

- Facilitate interoperability of PSS models
- Enable rapid PSS model development
- Provides common modeling pattern examples




* Work in progress

# Common Modeling Patterns

Interoperability of different IP PSS models

Different views of same address space with DDR controller example

IP PSS actions executed from different executors at SoC

Shared region between IP local memory and system memory

Grand Unified PSS (GUP) model ☺

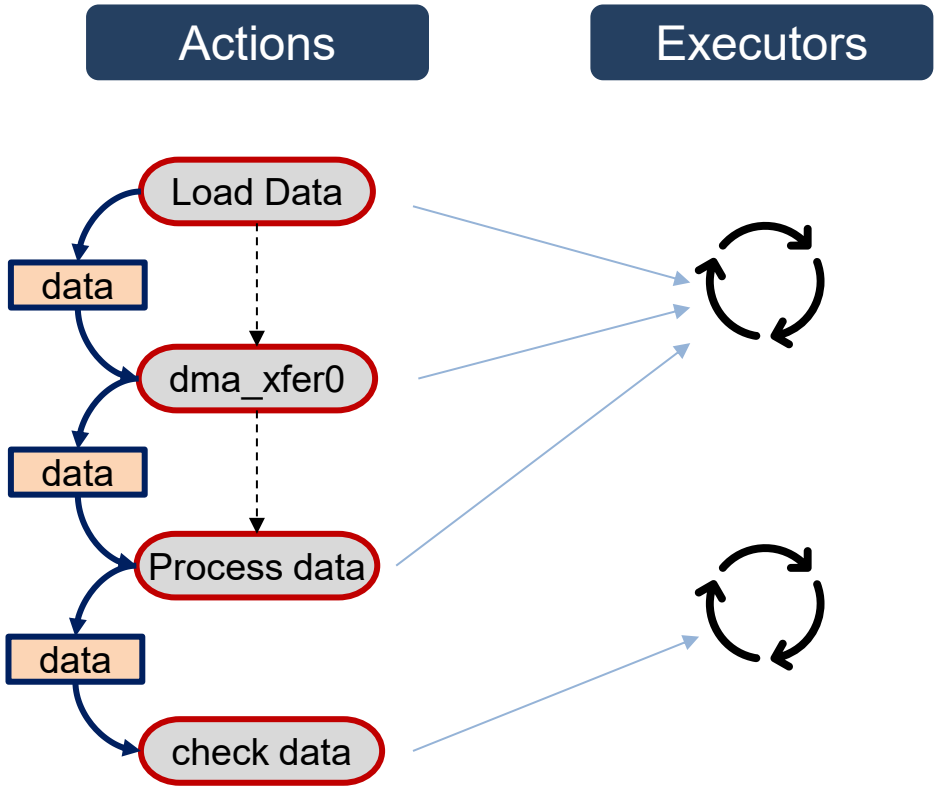
# PSS Methodology Library – PSSM*

* Work in progress

# The Executor

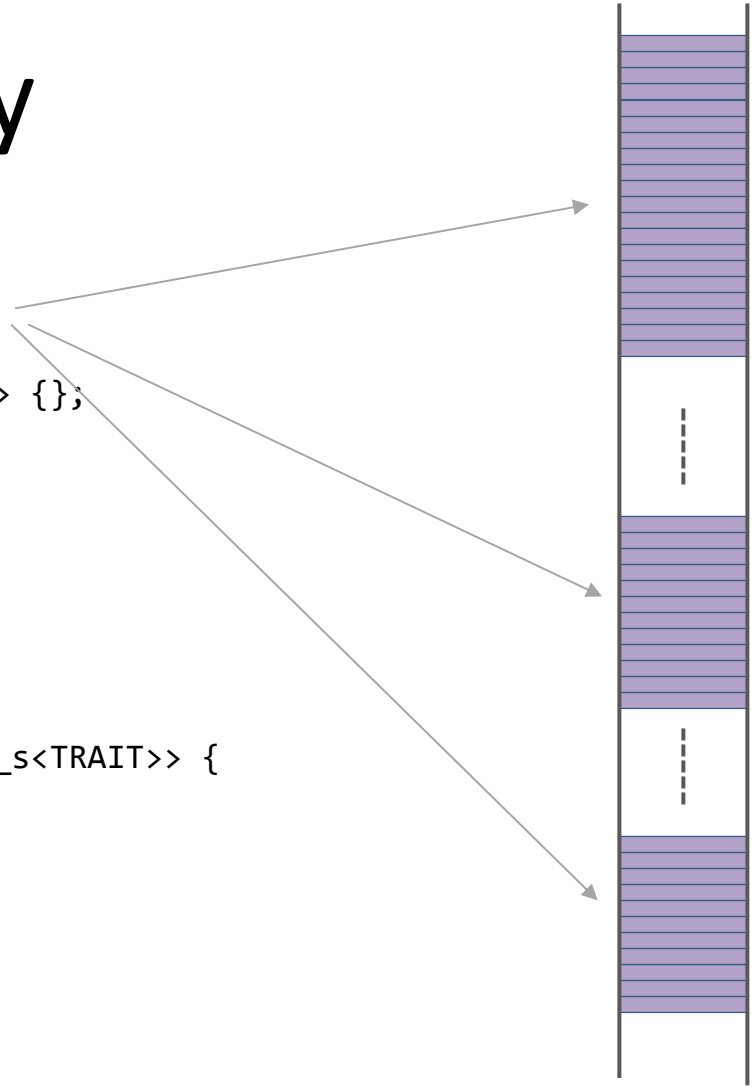
Agent of execution available to the test in the runtime environment

```
enum executor_tag_e {};

struct pssm_executor_trait_s : executor_trait_s
{
  rand executor_tag_e tag;
};
resource pssm_executor_r :
        executor_claim_s<pssm_executor_trait_s>
{
  rand executor_tag_e tag;
  constraint tag == trait.tag;
  constraint instance_id == (int)tag;
};
component pssm_executor_c :
         executor_c<pssm_executor_trait_s>
{
  executor_tag_e tag;
};
```



# The Memory

```
enum mem_block_e {};

struct mem_trait_s : addr_trait_s { rand mem_block_e mem_block;};

struct mem_segment_s<struct T=mem_trait_s> : transparent_addr_claim_s<T> {};
struct mem_segment_opaque_s<struct T=mem_trait_s> : addr_claim_s<T> {};

enum data_kind_e {raw_data};
struct data_s {
  rand data_kind_e kind;
  rand bit[64] size;
};
buffer data_buf <struct TRAIT = mem_trait_s, struct CLAIM = mem_segment_s<TRAIT>> {
  rand data_s data;
  rand CLAIM mem_seg;
  rand bit[64] size;
};
buffer data_buf_h  {
  rand data_s data;
  addr_handle_t mem_h;
};
```



# The Utilities

```
abstract action lock_executor {
     lock pssm_executor_r executor;
};

abstract action share_executor {
     share pssm_executor_r executor;
};

abstract action proc_context_a<pssm_pkg::executor_tag_e TAG> {
     constraint forall (it: pssm_pkg::lock_executor) {
          it.executor.tag == TAG;
     };
     constraint forall (it: pssm_pkg::share_executor) {
          it.executor.tag == TAG;
     };
};
```

Base action to lock an executor

Base action to share an executor

All actions that are derived form lock_executor or share_executor can be assigned an executor using proc_context_a action as base

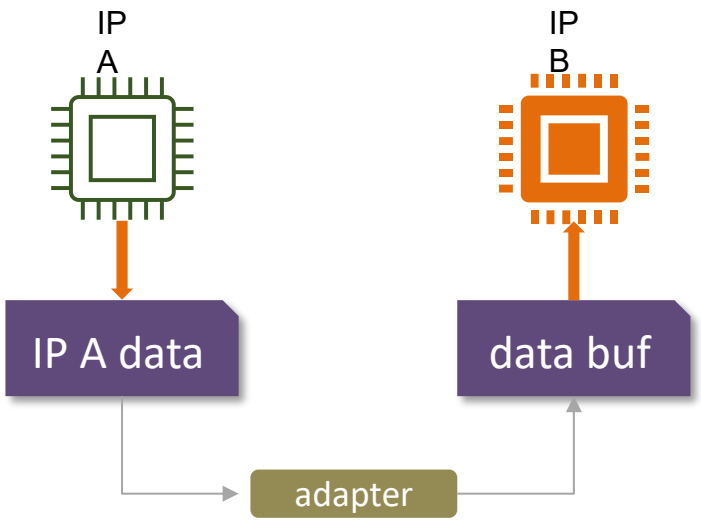
# Interoperability of Different IP PSS Models

IP A pkg

```
struct ip_A_data_s {
     rand int IP_A_attr_0;
     rand int IP_A_attr_1;
};
buffer data_buf : pssm_pkg::data_buf<> {
   rand ip_A_data_s ip_a_data;
};
```

IP A component

```
action data_buf_adapter : pssm_pkg::lock_executor {
   input data_buf data_buf_in;
  output pssm_pkg::data_buf_h data_buf_out;

  constraint data_buf_out.size == data_buf_in.size;
  exec post_solve {
    data_buf_out.mem_h =
    make_handle_from_claim(data_buf_in.mem_seg);
  };
};
action A_produce : pssm_pkg::lock_executor {
  output data_buf data_buf_out;
};
```




- Specialized IP data buffer automatically treated as generic buffer
- IPs provide an adapter action from specialized to generic buffers

# Interoperability of Different IP PSS Models

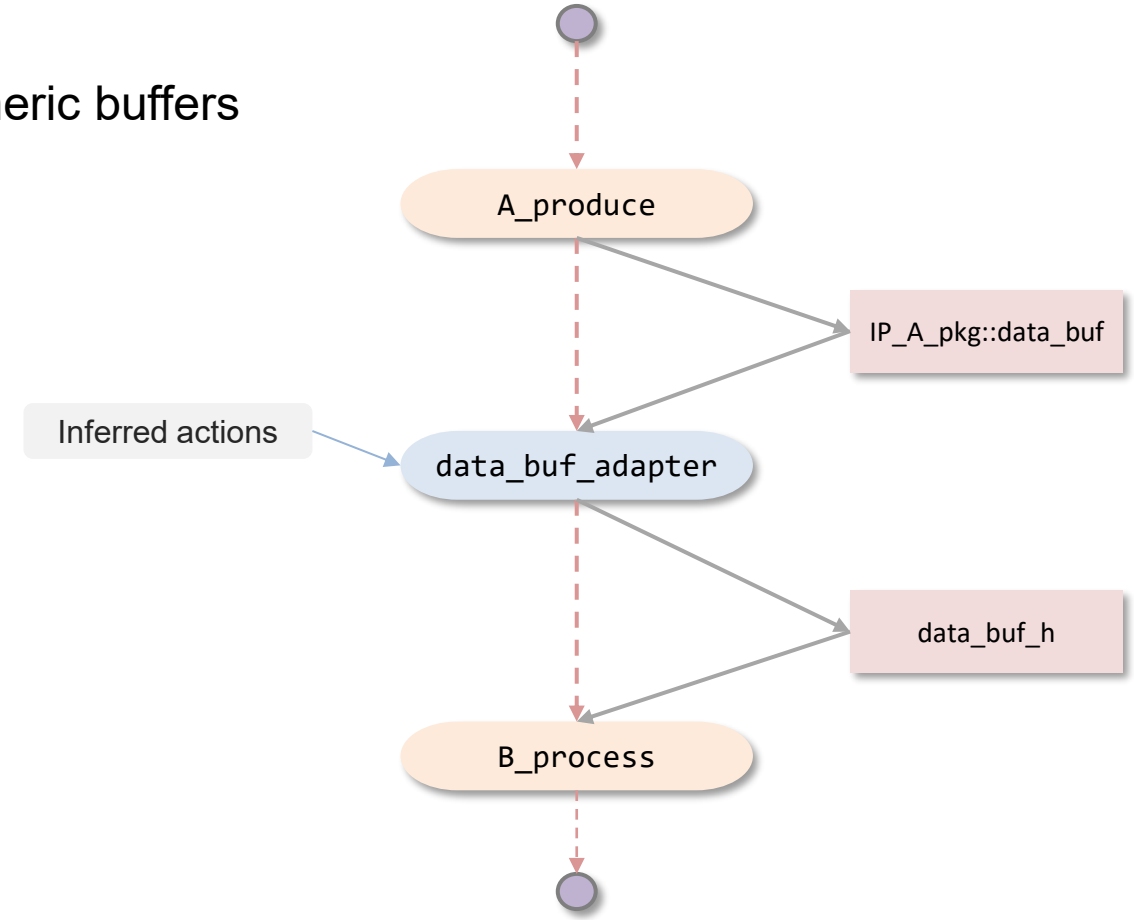
- IP B consume generic data buffer
- IP A adapter action facilitates conversion to generic buffers

IP B Component

```
 // consuming generic data buffer
action B_process : pssm_pkg::lock_executor {
  input data_buf_h data_buf_h_in;
};
```

Test

```
action {
  activity {
    do A_produce;
    do B_process;
  };
};
```



# Different Views into an Address Space - Memory Controller

- Memory controller IP tests look at memory as cacheline chunks with row, col, bank, etc. attributes
- IP provide an adapter action from mem ctrl buffer to generic buffers

Mem ctrl pkg

```
struct ddr_trait_s : addr_trait_s {
  rand bit[COL_HIGH - COL_LOW + 1] col;
  rand bit[BG_HIGH - BG_LOW + 1]  bg;
  rand bit[BANK_HIGH - BANK_LOW + 1]  bank;
  rand bit[ROW_HIGH - ROW_LOW + 1] row;
};

buffer data_buf : pssm_pkg::data_buf<ddr_trait_s> {
};
```

IP Component

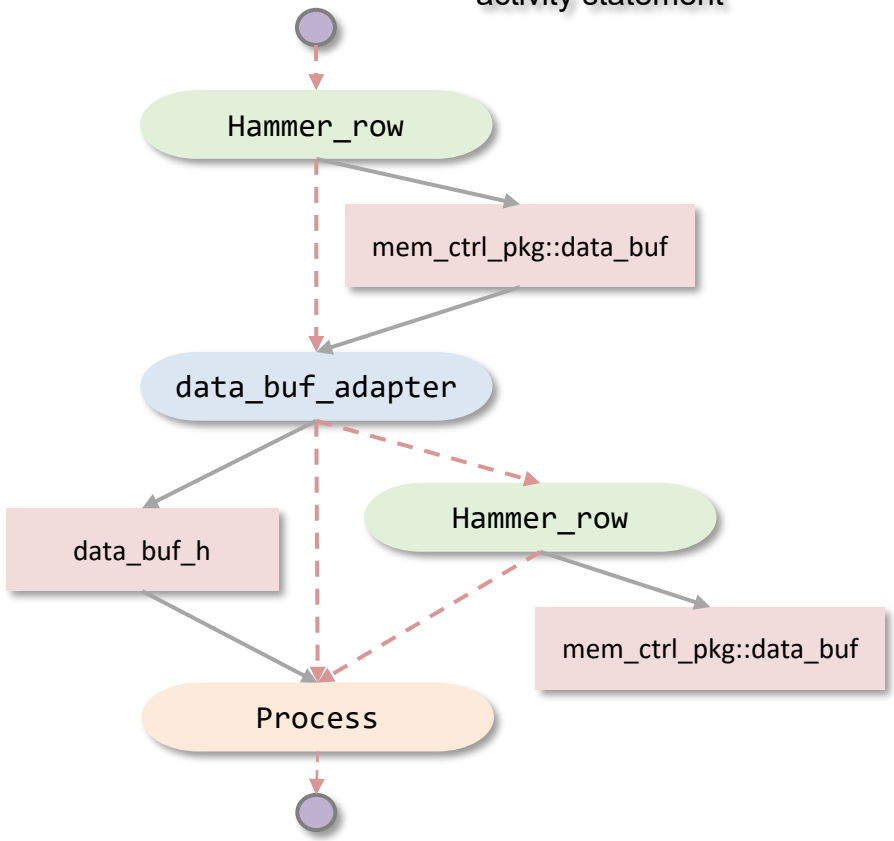
```
action data_buf_adapter : pssm_pkg::lock_executor {
  input data_buf data_buf_in;
  output pssm_pkg::data_buf_h data_buf_out;
  constraint data_buf_out.size == data_buf_in.size;

  exec post_solve {
    data_buf_out.mem_h = make_handle_from_claim(data_buf_in.mem_seg);
  };
};
action hammer_row : pssm_pkg::lock_executor {
  output data_buf data_buf_out;

  rand bit[ROW_HIGH - ROW_LOW + 1] row;
  rand int in [1..1024] num_trans;

  constraint data_buf_out.mem_seg.trait.row == row;
  constraint data_buf_out.size == CACHELINE_SIZE;
};
```

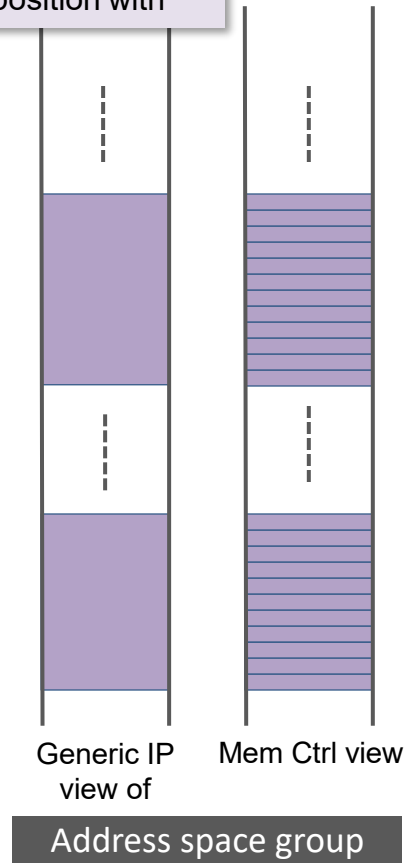
# Different Views into an Address Space - Memory Controller

SoC

```
extend struct ddr_trait_s {
  rand bit[MEMCTRL_HIGH - MEMCTRL_LOW + 1] mem_ctrl;
};
action hammer_mem_ctrl_row {
  rand bit[MEMCTRL_HIGH - MEMCTRL_LOW + 1] mem_ctrl;
  rand bit[ROW_HIGH - ROW_LOW + 1] row;
  rand int in [1..10] num_trans;
  activity {
    repeat(num_trans) {
    do mem_ctrl_c::hammer_row with {
        mem_ctrl== data_buf_out.mem_seg.trait.mem_ctrl;
        row == this.row;
        num_trans == this.num_trans;
      };
    };
  };
};
action complex_test {
  activity{
      do hammer_mem_ctrl_row;
     do process;
  };
};
```

Seemingly complex challenge or targeting memory controllers, row, col, bank etc., along with stimulus from other IPs becomes simple with PSS memory traits, address space group, and scenario composition with activity statement



# IP PSS Actions Executed from Different Executors at SoC

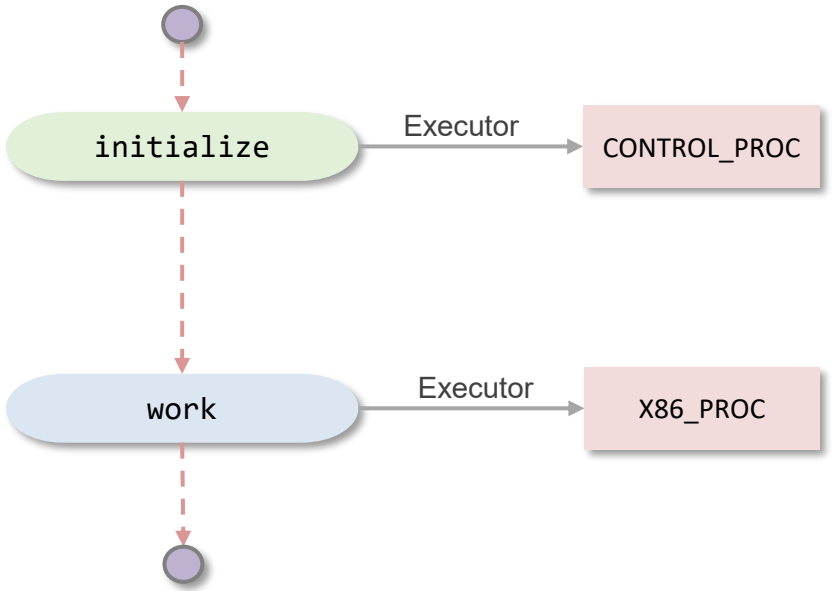
IP

```
component ip_A_c {
  action initialize : pssm_pkg::lock_executor {
  };
  action work : pssm_pkg::lock_executor {
  };
};
```

SoC

```
action initialize_soc : proc_context_a<CONTROL_PROC> {
   activity {
     atomic {
       do ip_A_c::initialize;
     }
   };
};
action test_ips_soc : proc_context_a<X86_PROC> {
   activity {
     do ip_A_c::work;
   };
};
action soc_test {
   activity {
     do initialize_soc;
     do test_ips_soc;
   };
};
```

- IP A is unaware of external executors
- IP A using PSSM executor
- SoC want to run certain IP A actions from a fixed executor
- SoC may run rest of the IP A actions from a set of executo



# Shared Region Between IP Local Memory and System Memory

## Information

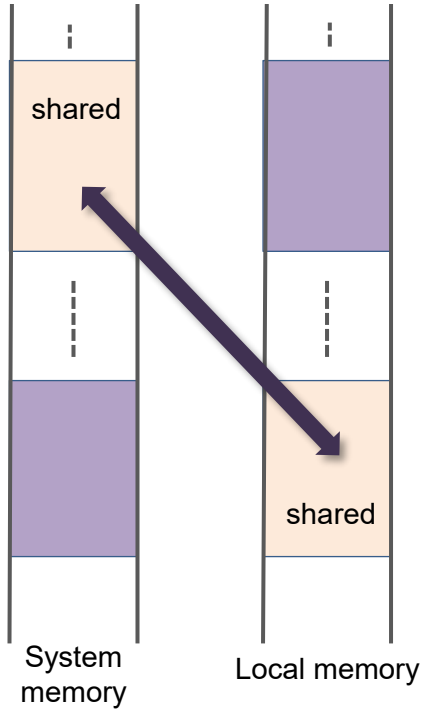
- IP contains a processor with local memory
- A region of local memory is mapped to system memory

## Goals

- Shared memory can be allocated by IP as well as any external IPs
  - A VIP in this example
  - Allocation needs to allocate space in both IP and system memory consistent with address translation scheme
- Address translation scheme might change mid-test

IP

```
enum ip_mem_block_e {SYS_MEM, LOCAL};

struct ip_mem_trait_s : addr_trait_s {
   rand ip_mem_block_e mem_block;
   constraint default mem_block == LOCAL;
};
buffer ip_data_buf :
          pssm_pkg::data_buf<ip_mem_trait_s> {
   constraint mem_seg.trait.mem_block == LOCAL;
};

// Shared buffer
buffer ip_sys_mem_data_buf : pssm_pkg::data_buf<> {
   addr_handle_t local_sys_mem_handle;
};

struct addr_translation_s {
   rand bit[54] sys_mem_block_size;
   rand bit[64] sys_mem_addr_base;
   addr_handle_t local_sys_mem_base_handle;
};

state addr_translation_state {
   rand addr_translation_s obj;
};
```




Two separate address spaces

# Shared Region Between IP Local Memory and System Memory

- Base action in package to handle address translation
- Every action that allocates memory in IP shared region should derive from this base action
- Base action uses state object to access translation information

IP

```
abstract action addr_translation_base_a {

   input addr_translation_state addr_translation_state_in;
   output ip_sys_mem_data_buf sys_mem_data_buf_out;

   constraint !addr_translation_state_in.initial;

   // Address translation
   exec pre_body {

     addr_handle_tsys_executor_mem_handle = make_handle_from_claim(sys_mem_data_buf_out.mem_seg);
     if(!addr_value_abs(sys_executor_mem_handle)) {
       std_pkg::fatal(1, "Absolute address not available for system memory")
     };
     bit[64] offset = addr_value_solve(sys_executor_mem_handle) - addr_translation_state_in.obj.sys_mem_addr_base;

     sys_mem_data_buf_out.local_sys_mem_handle = make_handle_from_handle(addr_translation_state_in.obj.local_sys_mem_base_handle, offset);
   };
};
```

# Shared Region Between IP Local Memory and System Memory

- IP and VIP actions that access shared region between two address spaces

IP

```
action sys_mem_write : addr_translation_base_a {

  lock ip_executor_r executor;

  rand bit[12] size;
  constraint sys_mem_data_buf_out.size == size;
  exec body{
    write32(sys_mem_data_buf_out.local_sys_mem_handle, 0xdeadbeef);
  };
};
```

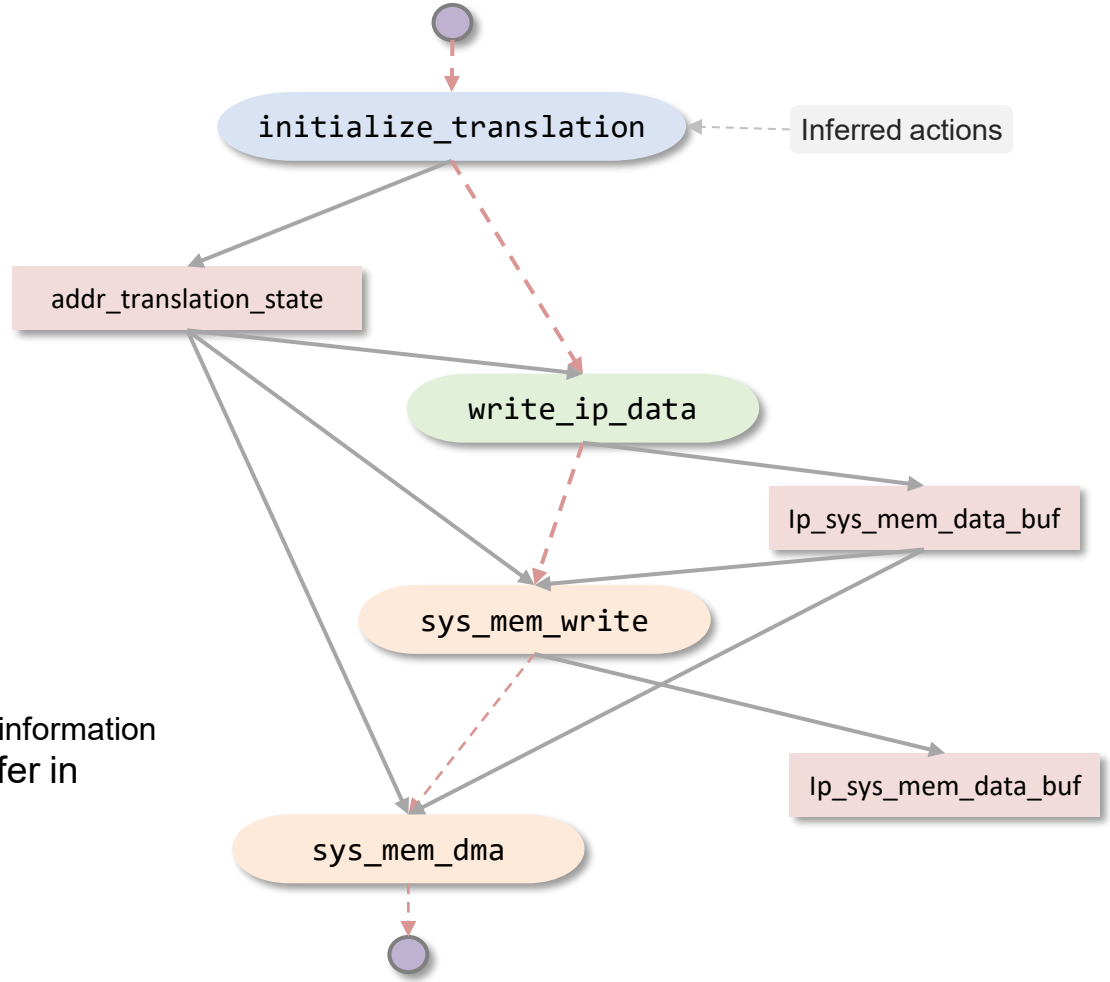
VIP

```
component vip_c {

  import pssm_pkg::*;
  import ip_pkg::*;

  action write_ip_data : addr_translation_base_a {

    lock pssm_executor_r executor;

    rand bit[12] size;
    constraint sys_mem_data_buf_out.mem_seg.size == size;
  };
};
```

# Shared Region Between IP Local Memory and System Memory

Test

```
action test {
   rand bit[64] size;

   constraint size == 64;

   activity {
      do vip_c::write_ip_data with {size == this.size;};
      do ip_c::sys_mem_write with {size == this.size;}
      do ip_c::sys_mem_dma;
   };
};
```




## Takeaway

- A buffer type for shared region between two address spaces
  - Shared buffer allocated in one address space
  - Translation is done using state object that keep dynamic translation information
- A base action in needed to do address translation on shared buffer in pre_body
- PSS 3.2 might provide a better mechanism

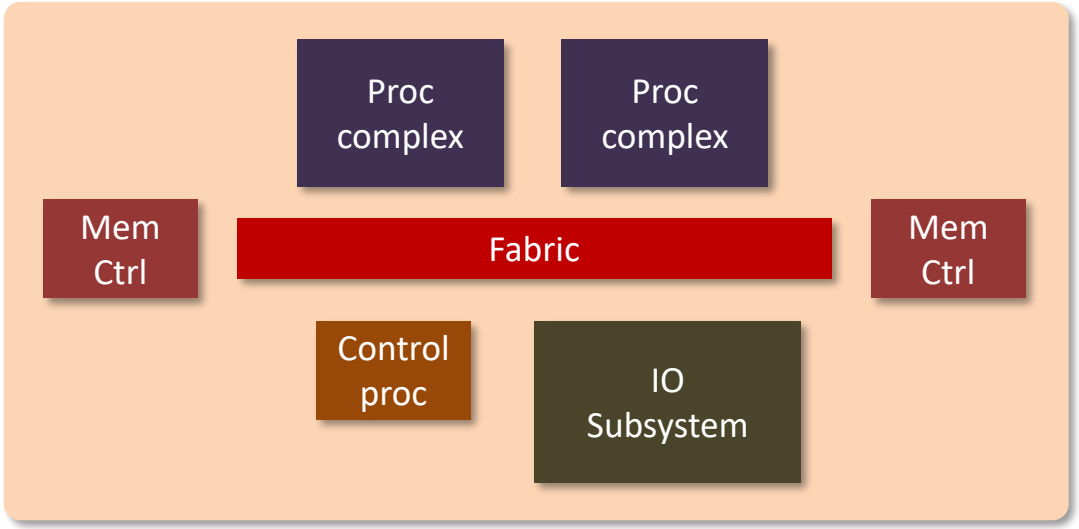
# Grand Unified PSS (GUP) model

**Setup**

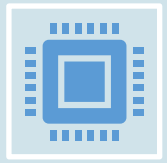
- IP PSS models imported
- Initialization and power sequencing rules created

**PSS Magic**

- Three IP functional test actions are solved
- Boot sequence generated for control processor
- IP local processor code generated
- Main processor code generated
- IP power sequencing inferred



# Conclusion



A simple PSS methodology speeds up complex multi-IP stimulus creation, improving pre- and post-silicon testing quality

PSSM offers standard solutions for common modeling issues, accelerating PSS model development

# PSS 3.1 PREVIEW

# PSS 3.1 Preview

- New PSS features generally fall into three categories
- Major new features
  - Behavioral coverage was the 3.0 major new feature
- Incremental feature enhancements
  - Expand the scope of an existing feature
- Improve LRM quality
  - Resolve ambiguities, fix typos, etc

# PSS 3.1 Preview

- Shared Memory Claims
  - PSS 3.0 supports unique memory claims
  - Example: False sharing – one mem region, many actions
  - Can implement by passing address handles today
  - Shared memory claims enable *modeling* the relationship

- Asynchronous Communication
  - Tests often need to wait for a condition to occur
  - Can poll, but interrupts are more efficient
  - A PSS model can now be notified of external events
  - Provides synchronization primitives to wait for events

Waits for IRQ

```
component dma_c {
  channel_c<int> irq_c;

  action m2m {
    // …
    exec body {
      // Start DMA transfer
      // …
      // Wait for DMA-done
interrupt
      irq_c.get();
    }
  }
}
```

# PSS 3.1 Preview: Generic Constraints

- Constraint blocks can become large and unwieldly
- Generic constraints enable reuse
  - Capture constraints and parameters
  - Allow reuse in multiple contexts
- Easier to develop and maintain constraints

Generic Constraint Declaration

Generic Constraint Instantiation

```
constraint move_up(
  array<cell_s, MAX_PATH_LEN> cells,
  cell_s cell, int t) {
     if(ROWS > 1) {
        cells[t+1].row == cell.row + 1;
        cells[t+1].col == cell.col;
     }
  }
}

// ...
constraint foreach(cell: cells[t]) {
  if(t < len) {
     // bottom left
     if(cell.row == 0 && cell.col == 0) {
        move_right(cells, cell, t);
        move_up(cells, cell, t);
     }
     // ...
```

# PSS 3.1 Preview: Initializing Sub-Actions

- **Random sub-action attributes can be constrained**
- **Can’t use constraints for non-rand attributes like address handles**
- **Inline initialization passes data to non-rand sub-action attributes**

```
action test {
  rand addr_claim_s<>            large_claim;

  activity {
    do sub1 {
      .hndl=make_handle_from_claim(large_claim, 0)} ;
    do sub2 {
      .hndl=make_handle_from_claim(large_claim, 0x10)};
  }
}
```

# PSS Comes of Age:
# Runtime Behavioral Coverage, Methodology and More

accellera SYSTEMS INITIATIVE™

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA