2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Portable Stimulus: What's Coming in 1.1 and What it Means For You

Portable Stimulus Working Group

accellera SYSTEMS INITIATIVE

2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

PSS 1.1 Tutorial Agenda

Introduction	• What is PSS • Abstract DMA model in PSS 1.0
Memory Allocation	• The problem • New PSS concepts
Higher-Level Scenarios	• The problem • New constructs
HSI Realization	• The problem • New concepts and constructs
System-Level Usage	• Portability • Complex scenarios
Conclusion	• Summary • What's next

- Tom Fitzpatrick, Mentor, a Siemens Business
- Prabhat Gupta, AMD
- Matan Vax, Cadence Design Systems
- Karthick Gururaj, Vayavya Labs
- Hillel Miller, Synopsys

Special Thanks to:
Dave Kelf, Breker Verification Systems
Josh Rensch, Semifore

accellera SYSTEMS INITIATIVE

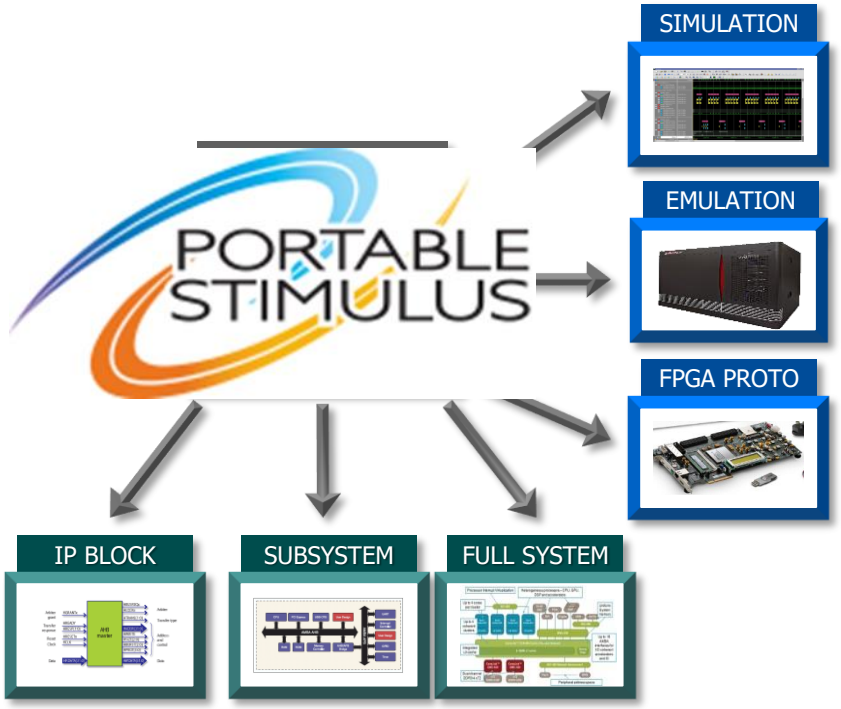
The Need for Verification Abstraction

Test content authoring represents major proportion of development

Disconnected cross-process methods

- Block
  - UVM tests laborious, error-prone
- SoC
  - Hard to hit corner-cases with C tests
- Post-Silicon
  - Disconnected diagnostic creation

Test portability, reuse, scaling, maintenance all problematic



Key Aspects of Portable Stimulus

Capture pure test intent

Partial scenario description

Composable scenarios

Formal representation of test space

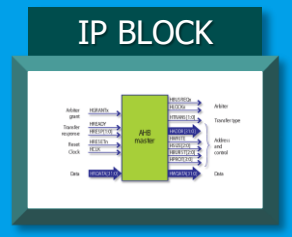
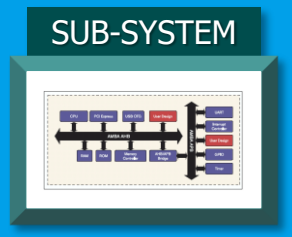
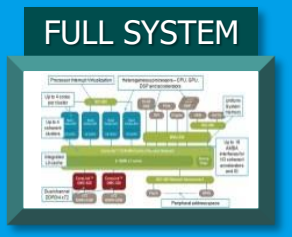
Automated test generation

Target multiple platforms

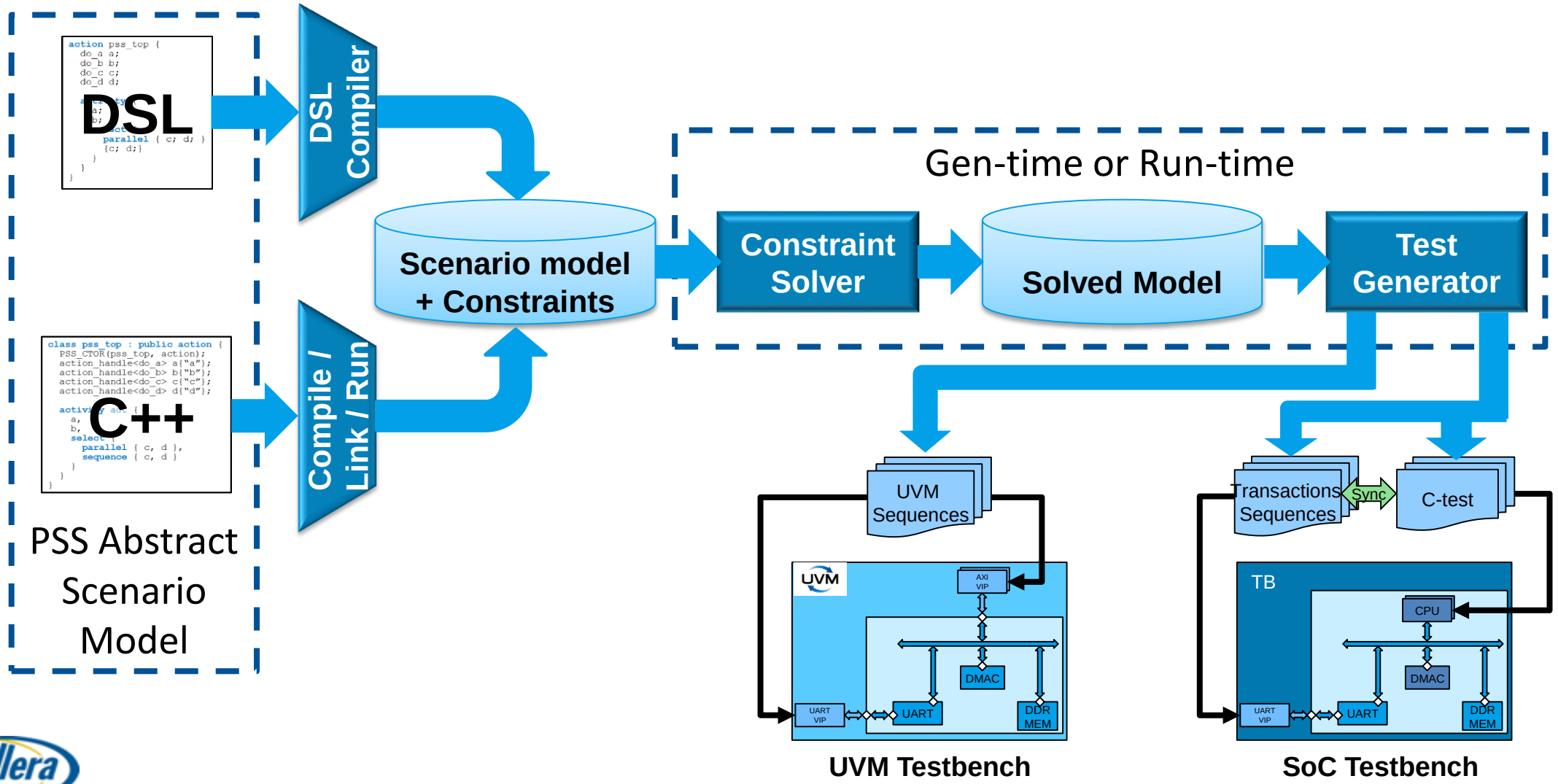
Separate test intent from implementation

High-coverage test generation across the verification process with much less effort

PSS Improves Individual Verification Phases

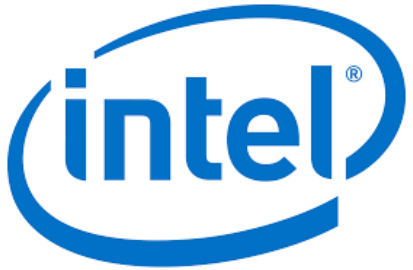
IP BLOCK	SUB-SYSTEM	FULL SYSTEM
		
Create block-level (UVM) tests & sequences based on scenario intent Easily compose complex, concurrent, high-coverage tests	Easily model system-level operations, reuse modular block level tests Drive hard-to-predict corner-case tests to flush out design operation	Generate software processor tests and IO transactions from common scenario model Flush out system functions with software-driven tests, while avoiding processor complexity

PSS Generalized Tool Flow



2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Active PSWG Participants



accellera SYSTEMS INITIATIVE

Key Additions to PSS 1.1

Modeling Improvements

- Better Reusable Content Capture
- Storage Allocation

Test Realization

- Register Space Definition
- Reusable Abstract Procedural Layer

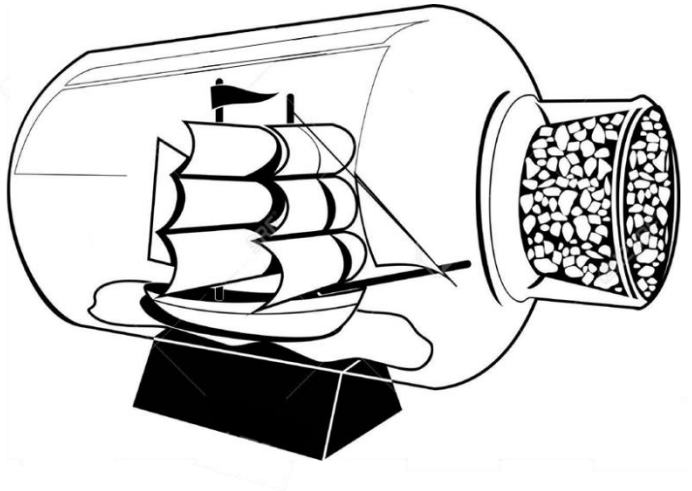
Programming Enhancements

- Templates
- Collection Types

What is a Portable Stimulus Model?

The Abstract Model

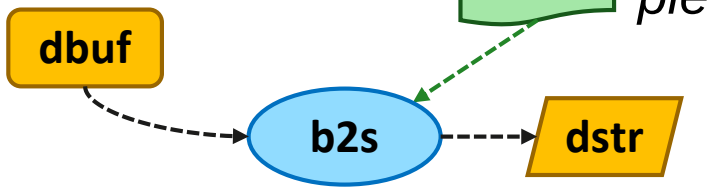
- *What* does it do



The Realization Layer

- *How* does it do what it does

The Parts of a PSS Model



resources represent *pieces of the system*

flow objects describe *communication*

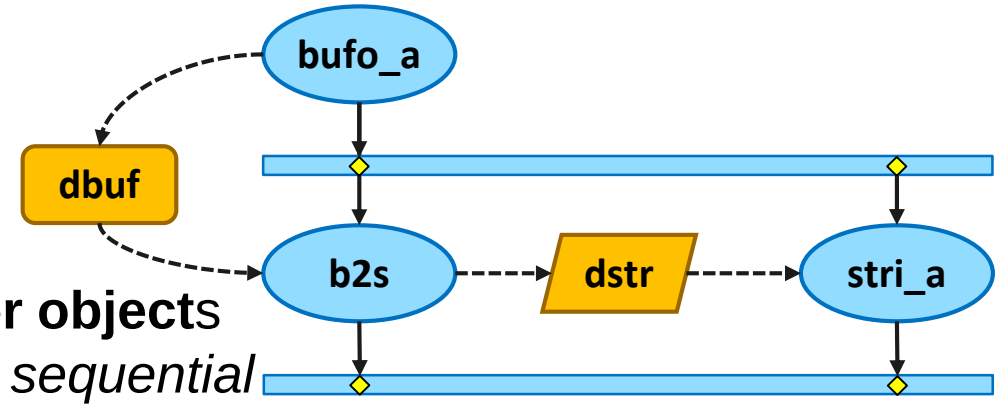
actions describe *behavior*

```
action b2s_a {
   input dbuf din;
   output dstr dout;

   rand int in [1..100] size;

   constraint din.size == size;

   lock channel_r chan;
}
```

actions may have *data fields*

actions may have *constraints*

Data Flow Objects Imply Scheduling



buffer objects imply *sequential*

stream objects imply *parallel*

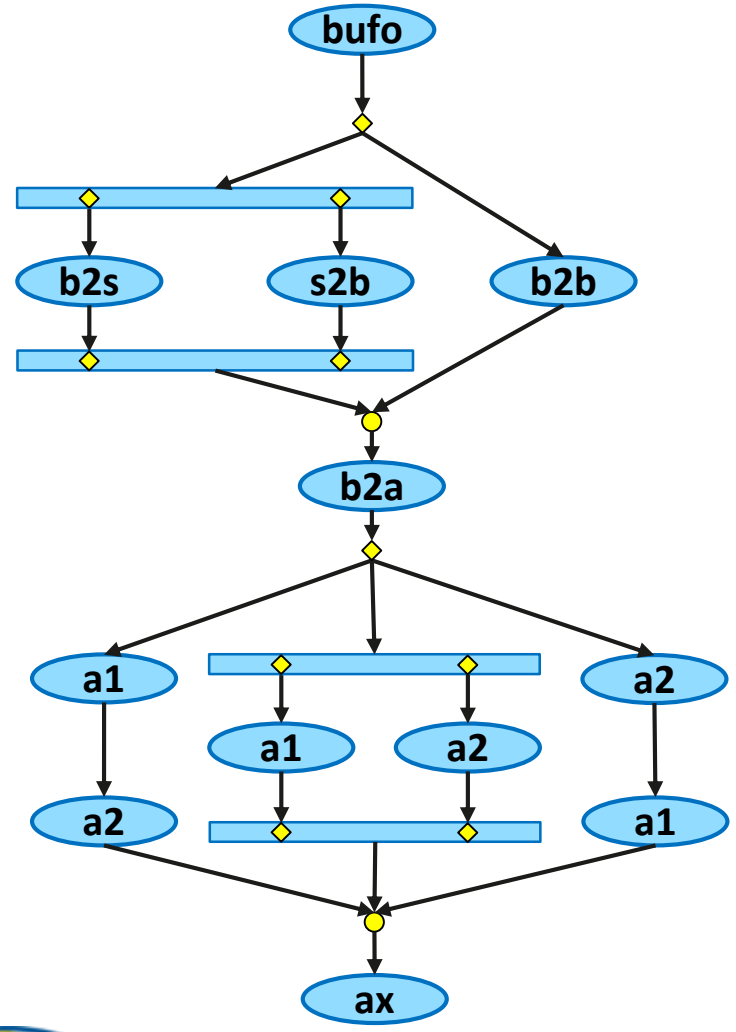
```
struct dat_s {
   rand bit[31:0] addr;
   rand int size;
}

buffer dbuf : dat_s {
   rand bit[12:0] key;
}

stream dstr : dat_s {
   rand bit dir;
}
```

activity describes *schedule*

```
action sched_a {
   bufo_a bufo;
   stri_a stri;
   b2s_a b2s;

   activity {
      bufo;
      parallel {
        b2s;
        stri;
      }
      bind bufo.dout b2s.din;
      bind b2s.dout stri.din;
   }
}
```

constraints on `bufo.dout` apply to `b2s.din`

Activities Can Define Flexible Scenarios

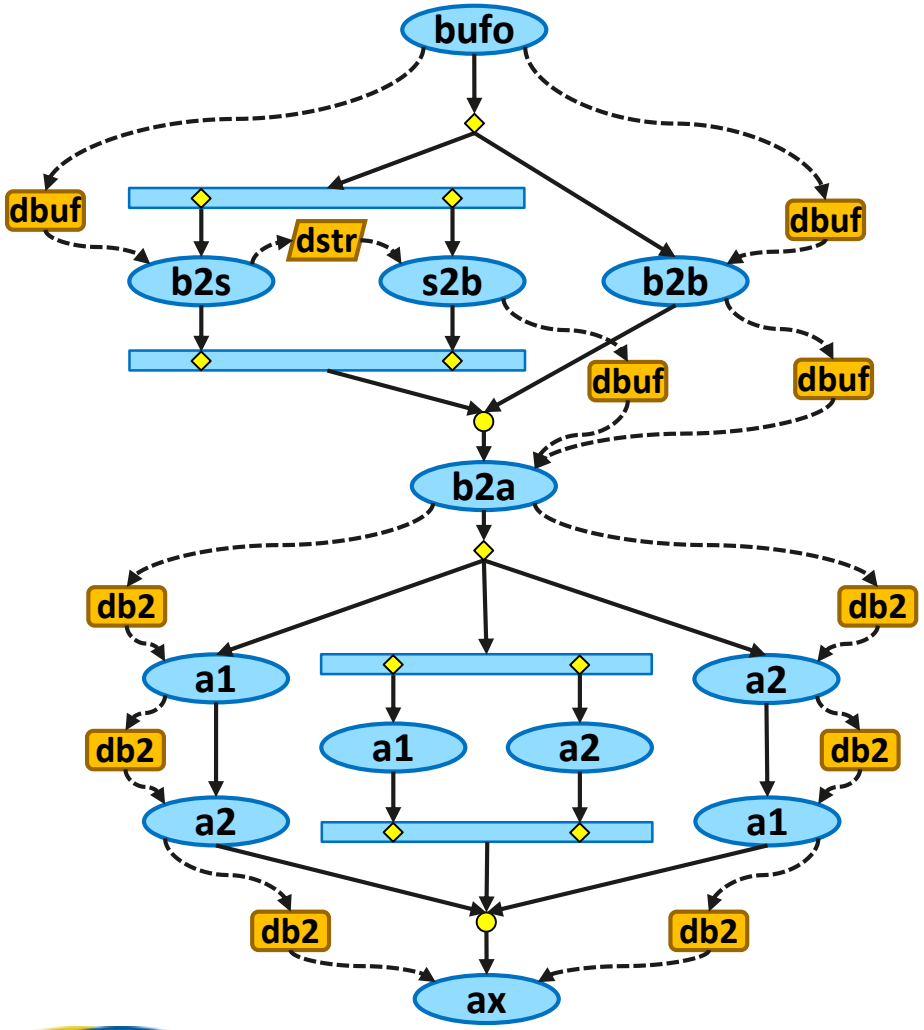


6 distinct control paths

Directed-Random
Scenario

```
action sched_a {
   bufo_a bufo;  s2b_a s2b; ax_a ax;
   b2s_a b2s; b2b_a b2b; my_a a1,a2;

   activity {
      bufo;
      select {
         parallel { b2s; s2b; }
         b2b;
      }
      b2a;
      schedule { a1; a2; }
      ax;
   }
}
```

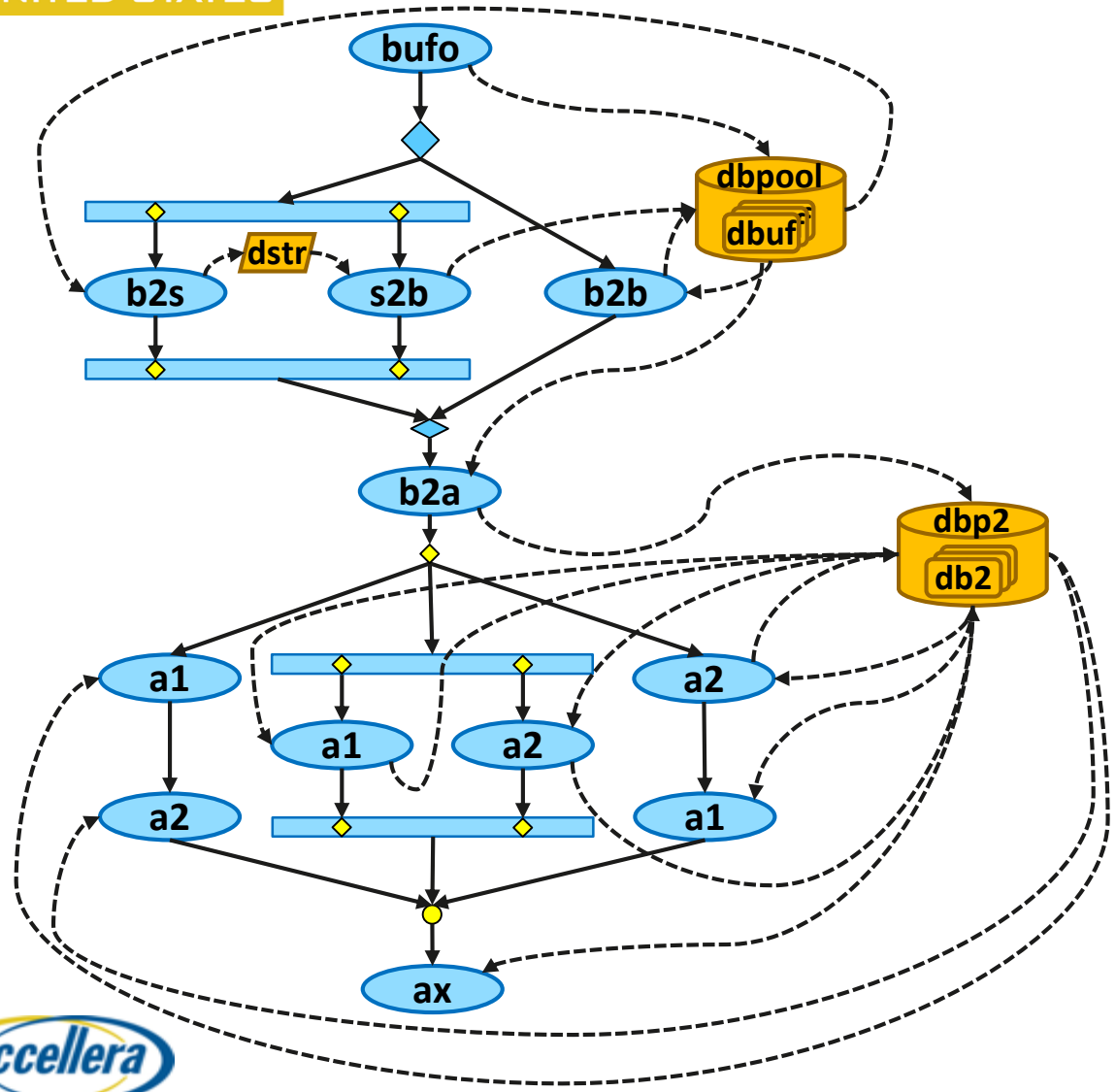
Activities Can Define Flexible Scenarios



Explicit binding
gets complicated

```
action sched_a {
   bufo_a bufo;  s2b_a s2b; ax_a ax;
   b2s_a b2s; b2b_a b2b; my_a a1,a2;

   activity {
      bufo;
      select {
         parallel { b2s; s2b; }
         b2b;
      }
      b2a;
      schedule { a1; a2; }
      ax;
   }
}
```

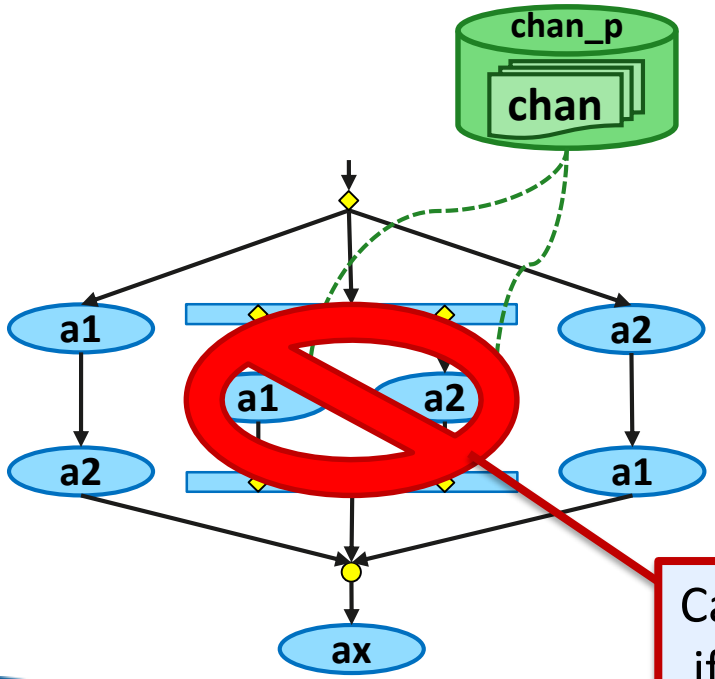
Flow Objects Managed via *Pools*




```
    pool dbuf dbpool_p;
    bind dbpool_p *;
    pool db2 dbp2_p;
    bind dbp2_p *;

action sched_a {
    bufo_a bufo;  s2b_a s2b; ax_a ax;
    b2s_a b2s; b2b_a b2b; my_a a1,a2;

    activity {
        bufo;
        select {
            parallel { b2s; s2b; }
            b2b;
        }
        b2a;
        schedule { a1; a2; }
        ax;
    }
}
```

Resources Add Dependencies

Resources may be ***lock***ed or ***share***d



Can't run in parallel if both require the same resource

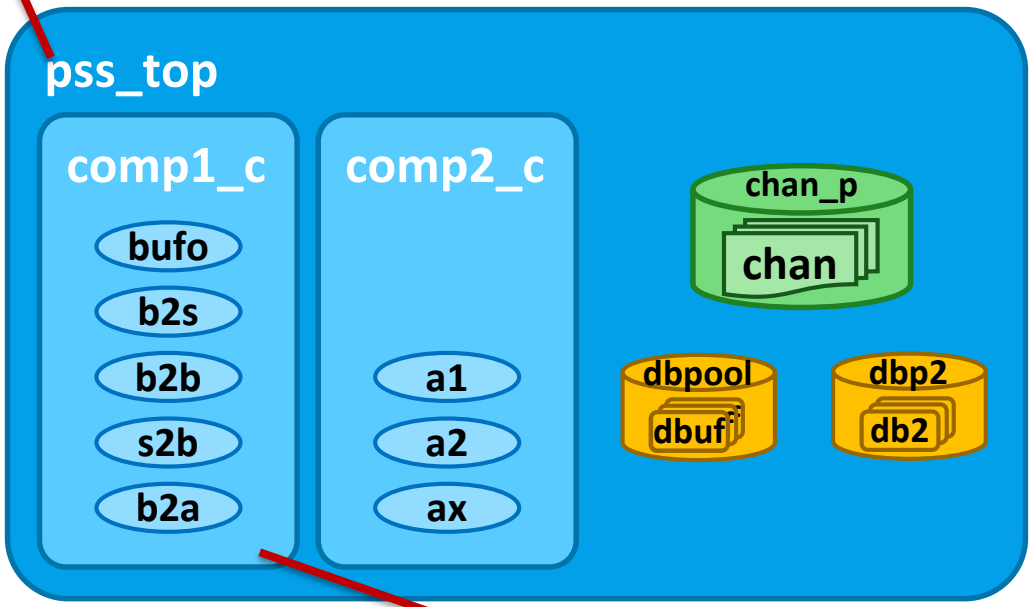
```
    pool [NUM] chan_r chan_p;
    bind chan_p *;

action sched_a {
    ...

    activity {
        schedule { a1; a2; }
        ax;
    }
}
```

Each resource in a pool is unique

```
resource chan_r : base_r {
    ...
}

action a1_a {
    ...
    lock chan_r chan;
}

action a2_a {
    ...
    lock chan_r chan;
}
```

Components Group Model Elements

Default top-level name



Components can instantiate subcomponents

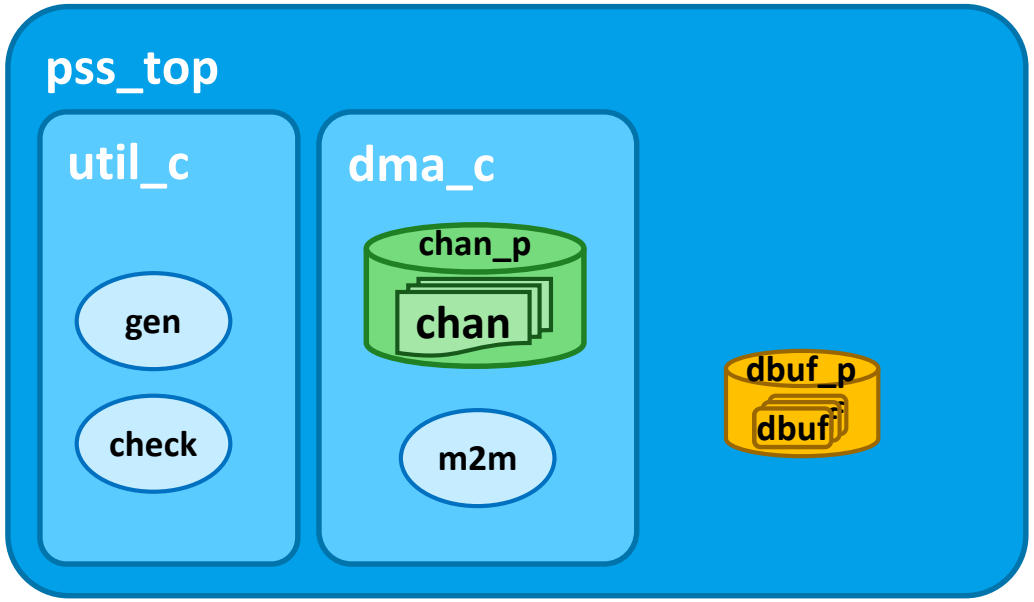
Pool binding is hierarchical

```
component pss_top {
  comp1_c c1;
  comp2_c c2;

  pool [NUM] chan_r chan_p;
  bind chan_p *;
  pool dbuf dbuf_p;
  bind dbuf_p *;
  pool db2 db2_p;
  bind db2_p *;
}

component comp2_c {
  my_a a1, a2;
  ax_a ax;

  activity {
    ...
  }
}
```

So What's the Point?

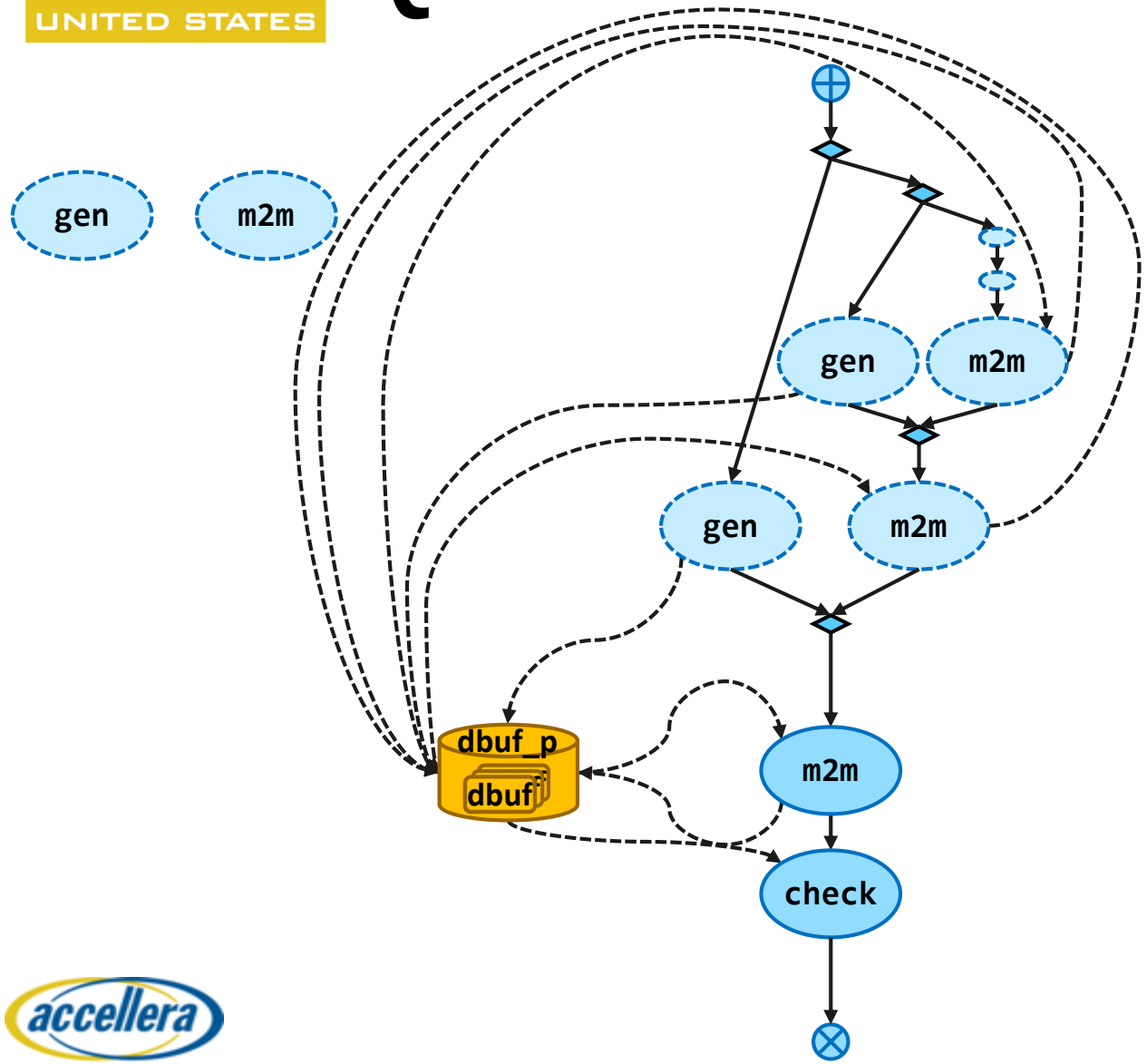
- Activity defines critical behaviors
  - May define *partial specification*
  - Tools will *infer* additional elements to complete the scenario
- Other parts of the model define how behaviors interact
  - Instantiated components define available actions
  - Flow object bindings constrain inference choices
  - Resources constrain scheduling options

Quick DMA Example



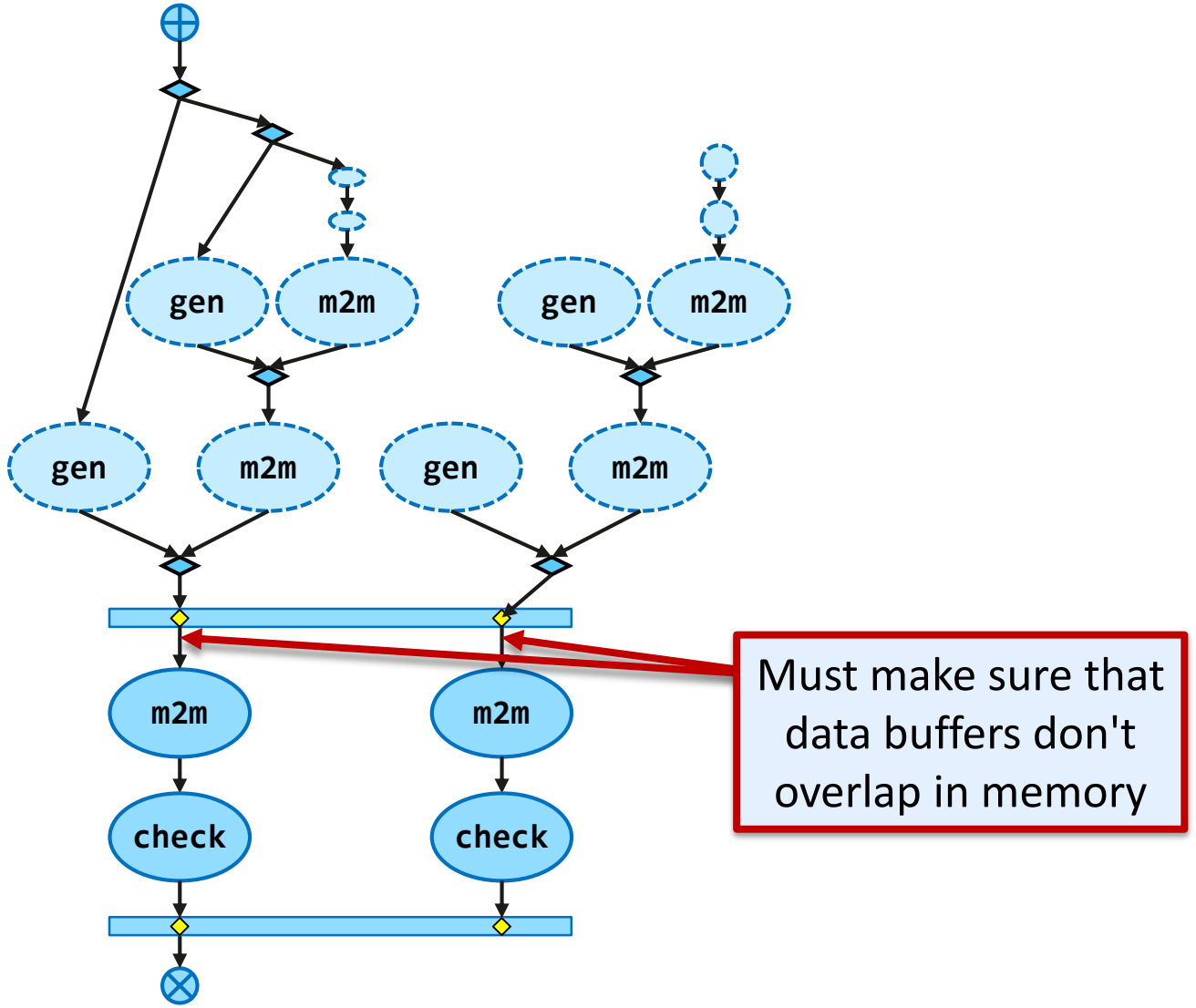

```
component dma_c {
  resource channel_r {};
  pool [NUM] channel_r chan_pool;
  bind chan_pool *;

  action mem2mem_xfer {
    input data_buff src_buff;
    output data_buff dst_buff;

    rand int in [1..100] size;

    constraint dst_buff.mem_seg.size == size;
    constraint dst_buff.mem_seg.size ==
                          src_buff.mem_seg.size;

    lock channel_r chan;
  }
}
```

Quick DMA Example




```
component pss_top {
  dma_c dma;
  util_c util;

  pool data_buff dbuf_p;
  bind dbuf_p *;

  action mem2mem_test {
    activity {
      do dmac::mem2mem_xfer;
      do util_c::check;
    }
  }
}
```

Quick DMA Example



PSS 1.1 Provides for Memory Allocation

Must make sure that data buffers don't overlap in memory

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

Target Template *Exec* Blocks

```
extend action mem2mem_xfer {
  exec SV """
    begin
      m2m_xfer xseq = m2m_xfer::type_id::create("xseq");
      xseq.item.src == {{src_buff.addr}};
      xseq.item.dst == {{dst_buff.addr}};
      xseq.item.size == {{size}};
      xseq.start(m_env.m_agent.m_seqr);
    end
  """
}
```

Define SV implementation

Pass PSS args

```
extend action mem2mem_xfer {
  exec C """
    SRC_REG.write({{src_buff.addr}});
    DST_REG.write({{dst_buff.addr}});
    SZ_REG.write({{size}});
    CSR_REG.write(GO);
  """
}
```

Define C implementation

Procedural Interface *Exec* Blocks

```
package func_pkg {
    function void m2m_xfer(bit[31:0]src,
                           bit[31:0]dst,
                           bit[6:0]size);
}
```

Single method prototype

```
extend action mem2mem_xfer {
  import func_pkg::*;
  exec body {
    m2m_xfer(src_buff.addr,
             dst_buff.addr,
             size);
  }
}
```

Must provide implementations in both SV and C

This is why PSS1.1 adds an abstract procedural layer

MEMORY ALLOCATION

Memory Allocation Topics

Problem

Why model allocation

Concepts

Address space

Address space region

Allocation claim

Address space handle

Application

Address space operations

Packed struct

Descriptor chain

Portability and controllability problem

IP testbench

Need to manage memory resource

```
src_addr = mem_mgr.allocate(size, properties);
dst_addr = mem_mgr.allocate(size, properties);

// Program descriptor in memory
// ...

// Program IP
write_reg(CTRL_REG, descriptor)

mem_mgr.deallocate(src_addr);
mem_mgr.deallocate(dst_addr);
```

HOW TO ESTABLISH INTERESTING RELATIONSHIP BETWEEN SOURCE AND DESTINATION

WILL TEST RUN OUT OF MEMORY

IS MEMORY MANAGER PORTABLE

Portability and controllability problem

Sub-system

Reconcile memory management of different IP

```
// Manage two different memory manager
addr1 = IP1_mem_mgr.allocate(size, properties);
addr2 = IP2_mem_mgr.allocate(size, properties);

//-------------------- OR ----------------------
//
// Create unified memory manager
addr1 = unified_mem_mgr.allocate(size, properties);
addr2 = unified_mem_mgr.allocate(size, properties);
```

ESTABLISH INTERESTING RELATIONSHIP BETWEEN ADDR1 AND ADDR2

ARE MEM MANAGERS COMPATIBLE

IS MEMORY MANAGER PORTABLE

Model allocation – portability problem

SoC and post-silicon

Reuse stimulus that allocate and access memory

```
// Processor based tests

// hardcode addresses
Uint64_t addr = 0x123456;

// Simple mem manager at runtime
addr = malloc(0x1000);

// How to create test that would hit all memory
controller?

// IP tests need to be re-written
```

HARDCODE ADDRESS OR COSTLY ALLOCATOR

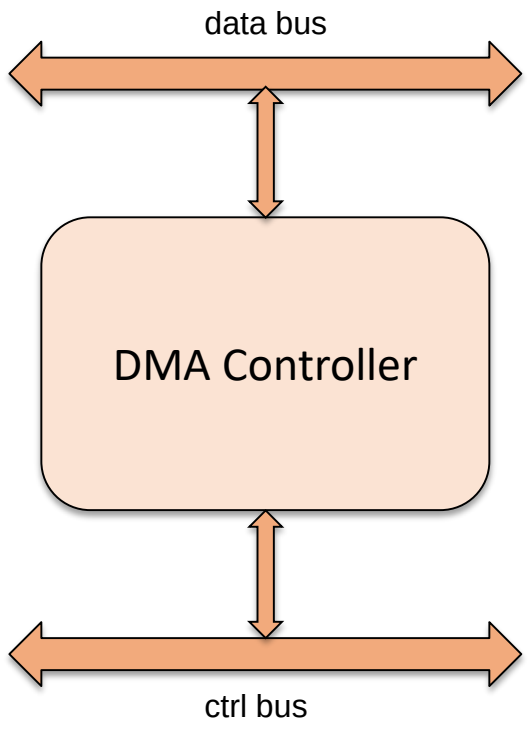
IS TEST PORTABLE

IS MEMORY MANAGER PORTABLE

DMA IP – PSS 1.0

Test space

DMA channels
DMA operation
Modes
DMA Size




```
buffer data_buff {
    rand bit[32] addr;
};

component dma_c {

   resource channel_r {};

   action mem2mem_xfer {
      input  data_buff src_buff;
      output data_buff dst_buff;

      rand int in [1..100] size;
      lock channel_r chan;
   };

   pool [NUM_DMA_CHANNELS] channel_r chan_pool;
   bind chan_pool *;
};
```

Memory allocation – PSS issues

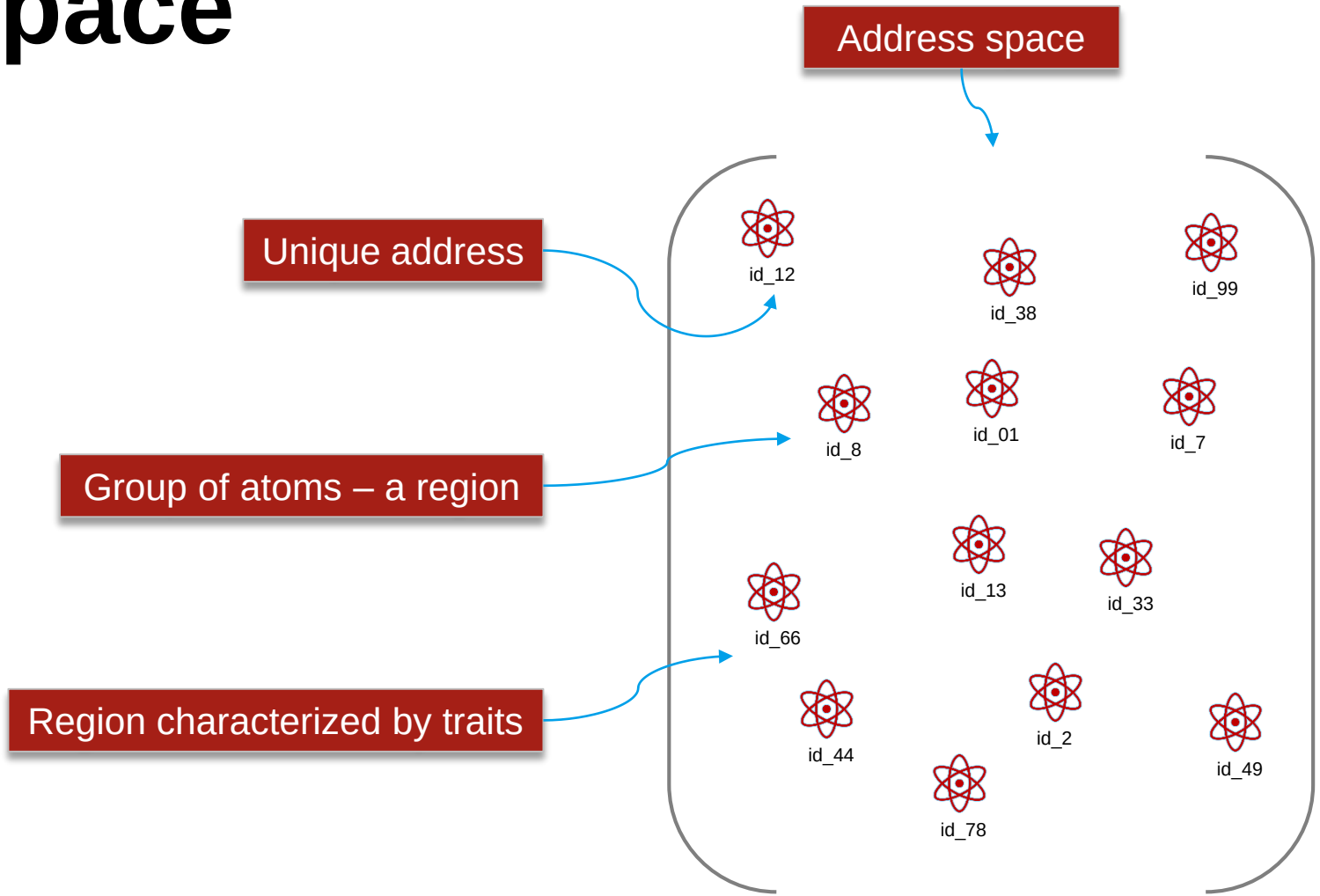
- Consistent allocation across randomly scheduled concurrent actions
  - PSS tool needs to know allocation lifetimes
- Test intent to model interesting address corner cases
  - Hard coding addresses doesn't work
- Runtime allocator works but
  - Not portable, slow
  - Not enough controllability to achieve test indent

Difficult problem at SoC where PSS tool is allocating memory at solve time

Address space

Address space is a space of ***atoms*** accessible using ***unique addresses***. Agents in system can allocate and access one or more atoms from this space. An address space is characterized by its properties called ***traits*** and ***primitive operations*** over it.

An address space may be composed of ***regions***. Regions could be allocatable or non-allocatable. Allocatable regions are characterized by specific value of space ***trait***.



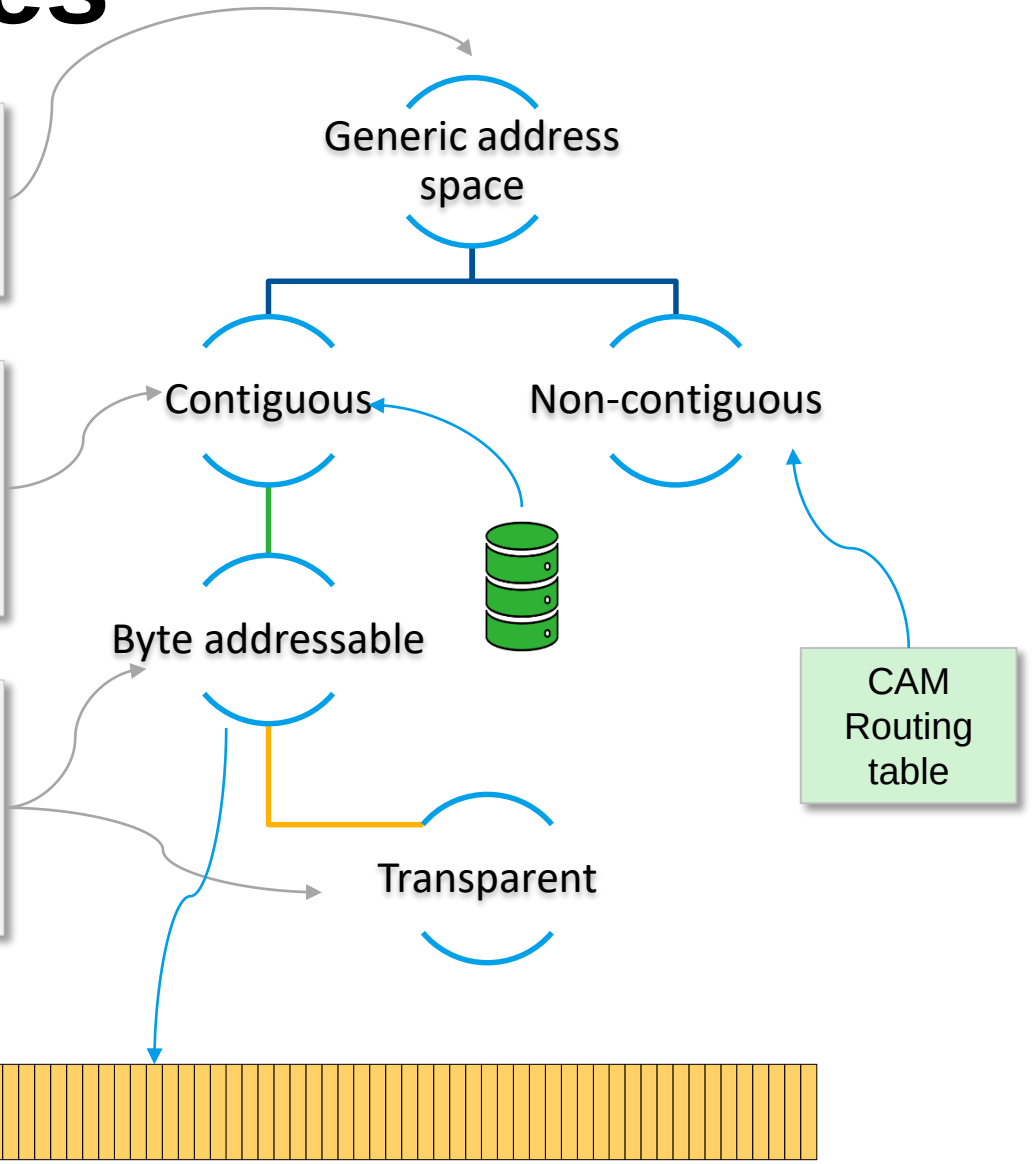
Common case of address space is byte addressable system memory

Address space types

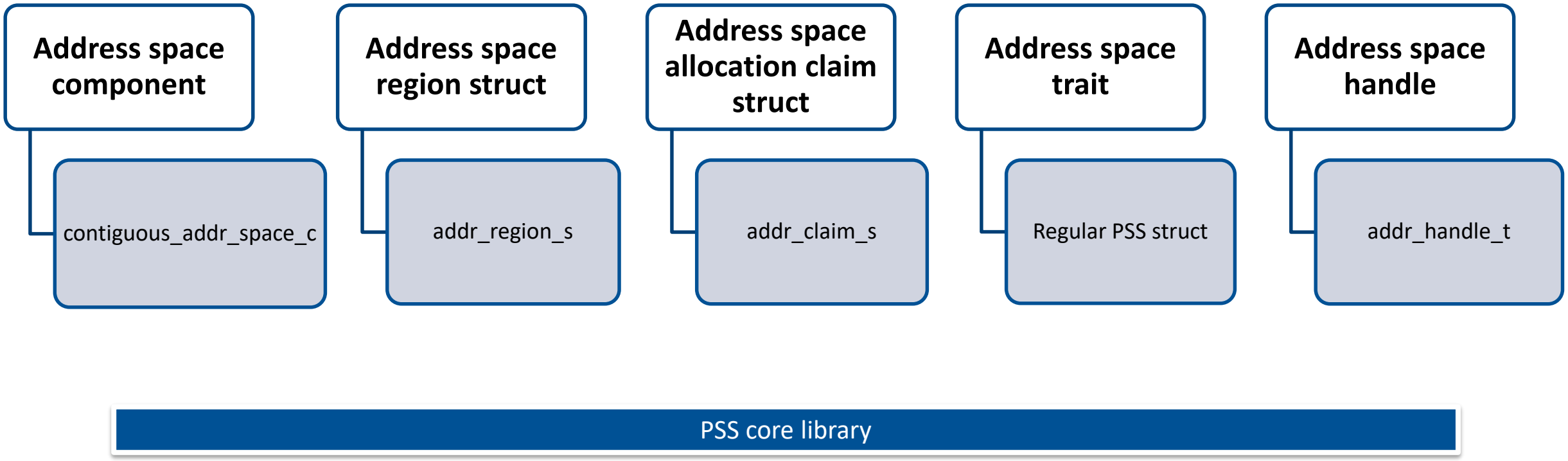
A set of storage atoms accessible using unique addresses. Agents in a system may allocate one or more atoms for the exclusive use of actions/objects

An address space whose addresses are non-negative integer values. This space is defined as containing atoms that are contiguously addressed by non-negative integer values. Multiple atoms can be allocated in one contiguous chunk

A contiguous address space whose storage atom is a byte, and to/from which PSS data can be written/read using standard generic operations. PSS core library standardize generic APIs to write data to or read data from any address value as byte



Address space APIs



Address space component

```
component contiguous_addr_space_c<struct TRAIT = null_trait_s>
     : addr_space_base_c
  {
     function void add_region(addr_region_s<TRAIT> r);

     function void add_nonallocatable_region(addr_region_s<> r);

     bool byte_addressable = true;
  };

component transparent_addr_space_c<struct TRAIT = null_trait_s>
    : contiguous_space_c<TRAIT>
  {
     // It is illegal to pass a non-transparent region to the
     // add_region() function.
  };
```

PSS library

```
extend component pss_top {

   my_ip ip;

   transparent_addr_space_c<>   sys_mem;
   transparent_addr_space_c<> local_mem;

   exec init {

      transparent_addr_region_s<> r0;
      transparent_addr_region_s<> r1;

        sys_mem.add_region(r0);
      local_mem.add_region(r1);
   }
}
```

PSS user code

Address space trait

```
extend component pss_top {

   my_ip ip;

   transparent_addr_space_c<mem_trait_s>  sys_mem;
   transparent_addr_space_c< fb_trait_s>  local_mem;

   exec init {
      transparent_addr_region_s<> r0;
      transparent_addr_region_s<> r1;

      r0.size           = 1024;
      r0.base_address   = 4096;

      r0.trait.ctype     = WB;
      r0.trait.sec_level = level0;

      r1.size           = 1024;
      r1.base_address   = 0;

        sys_mem.add_region(r0);
      local_mem.add_region(r1);
   }
}
```

```
struct mem_traits_s {
    rand cache_attr_e      ctype;
    rand security_level_e  sec_level;
}
struct fb_trait_s {}
```

- Address space trait is fixed size struct to describe properties of allocatable regions of an address space.
- Allocation trait is used by a PSS tool to define regions as well as match allocation claims to allocatable regions of an address space.

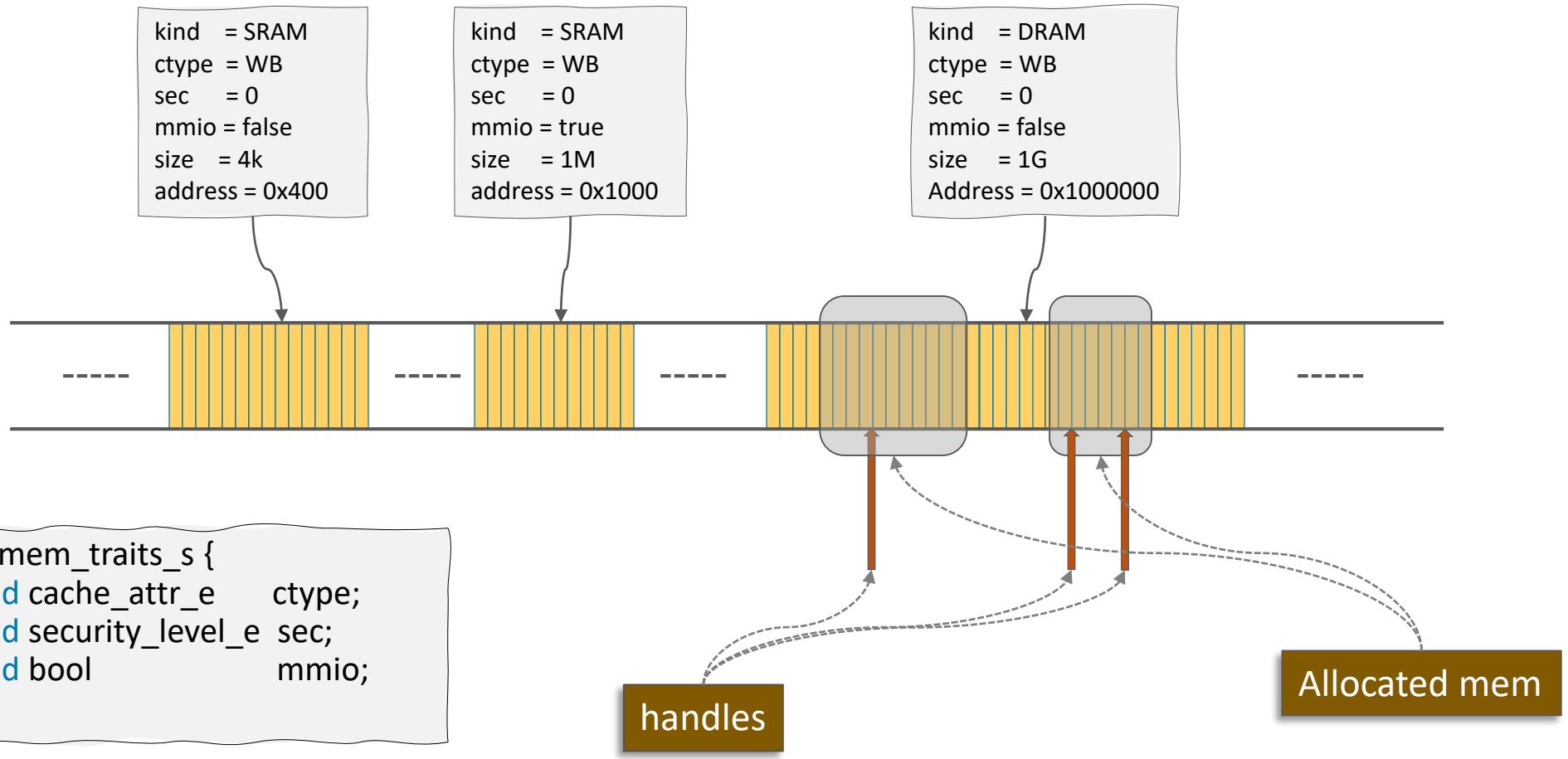
Address space region

```
struct addr_region_base_s {
   bit[64] size;
};

struct addr_region_s <struct TRAIT = null_trait_s>
      : addr_region_base_s
{
    TRAIT   trait;
};

struct transparent_addr_region_s <struct TRAIT = null_trait_s>
      : addr_region_s<TRAIT>
{
   bit[64]    addr;
};
```

An address space is composed of region(s). Region(s) map to parts of an address space. A region is characterized by address space ***trait value***

An allocation claim would use trait to map to a region from address space. Regions with trait values that satisfy the claim's trait constraints are the candidates matching regions

Byte addressable memory




```
struct mem_traits_s {
    rand cache_attr_e      ctype;
    rand security_level_e  sec;
    rand bool              mmio;
}
```

Allocate memory - claims

- Instance of claim struct addr_claim_s allocates memory
- Opaque claim – not reasoning about absolute resolved address for allocation
- Lifetime of allocation is same as object that creates claim struct instance
- Lifetime of allocation can be extended with address space handles
- Transparent claim can reason about absolute memory address

```
struct mem_trait_s { };

component dma_c {

  action mem2mem_xfer {

    rand addr_space_pkg::addr_claim_s<mem_trait_s>    src_mem;
    rand addr_space_pkg::addr_claim_s<mem_trait_s>    dst_mem;

    constraint src_mem.size == dst_mem.size;

    rand int in [1..100] size;

    constraint size == src_mem.size;
  };
};
```

Unique allocation

Allocate memory - claims

- Instance of claim struct addr_claim_s allocates memory
- Opaque claim – not reasoning about absolute resolved address for allocation
- Lifetime of allocation is same as object that creates claim struct instance
- Lifetime of allocation can be extended with address space handles
- Transparent claim can reason about absolute memory address

```
struct mem_trait_s { };

buffer data_buff {
  rand addr_space_pkg::addr_claim_s<mem_trait_s> mem_seg;
};

component dma_c {

  action mem2mem_xfer {

    input  data_buff  src_buff;
    output data_buff  dst_buff;

    constraint dst_buff.mem_seg.size == src_buff.mem_seg.size;

    rand int in [1..100] size;
    lock channel_r chan;
  };
};
```

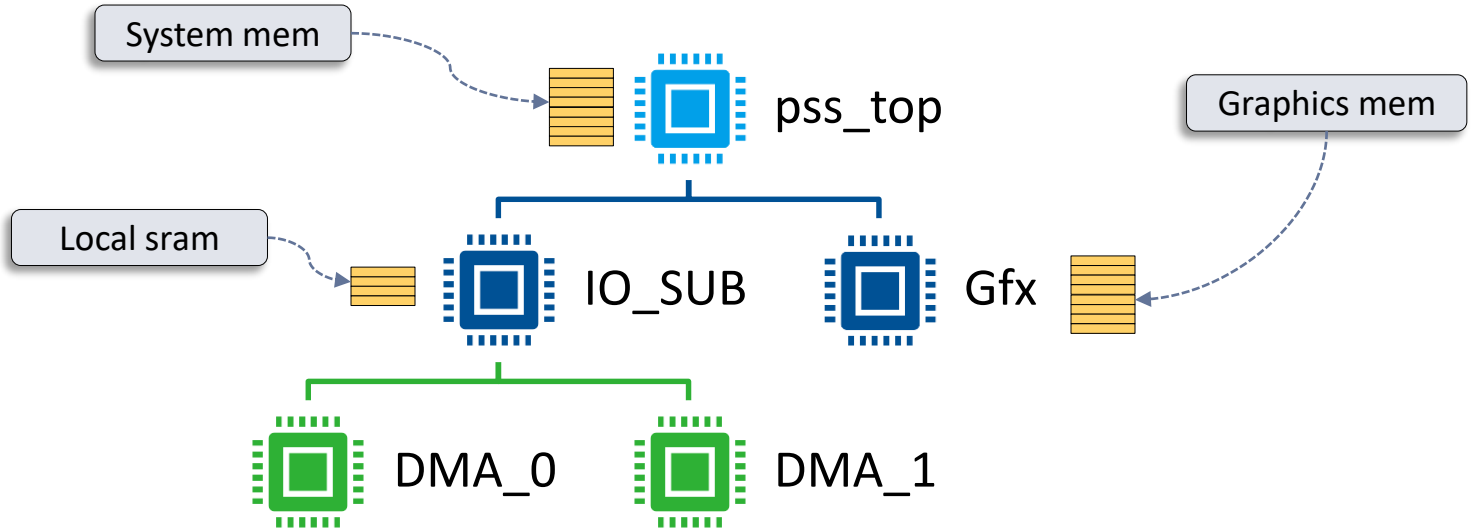
Allocate memory - claims

- Instance of claim struct addr_claim_s allocates memory
- Opaque claim – not reasoning about absolute resolved address for allocation
- Lifetime of allocation is same as object that creates claim struct instance
- Lifetime of allocation can be extended with address space handles
- Transparent claim can reason about absolute memory address

```
extend component pss_top {

  dma_c dma_0;
  dma_c dma_1;

  contiguous_addr_space_c<mem_trait_s> mem_addr_space;

  addr_region_s<mem_trait_s> sram_region;

  exec init {
    sram_region.trait.kind = SRAM;

    mem_addr_space.add_region(sram_region);
  }
```

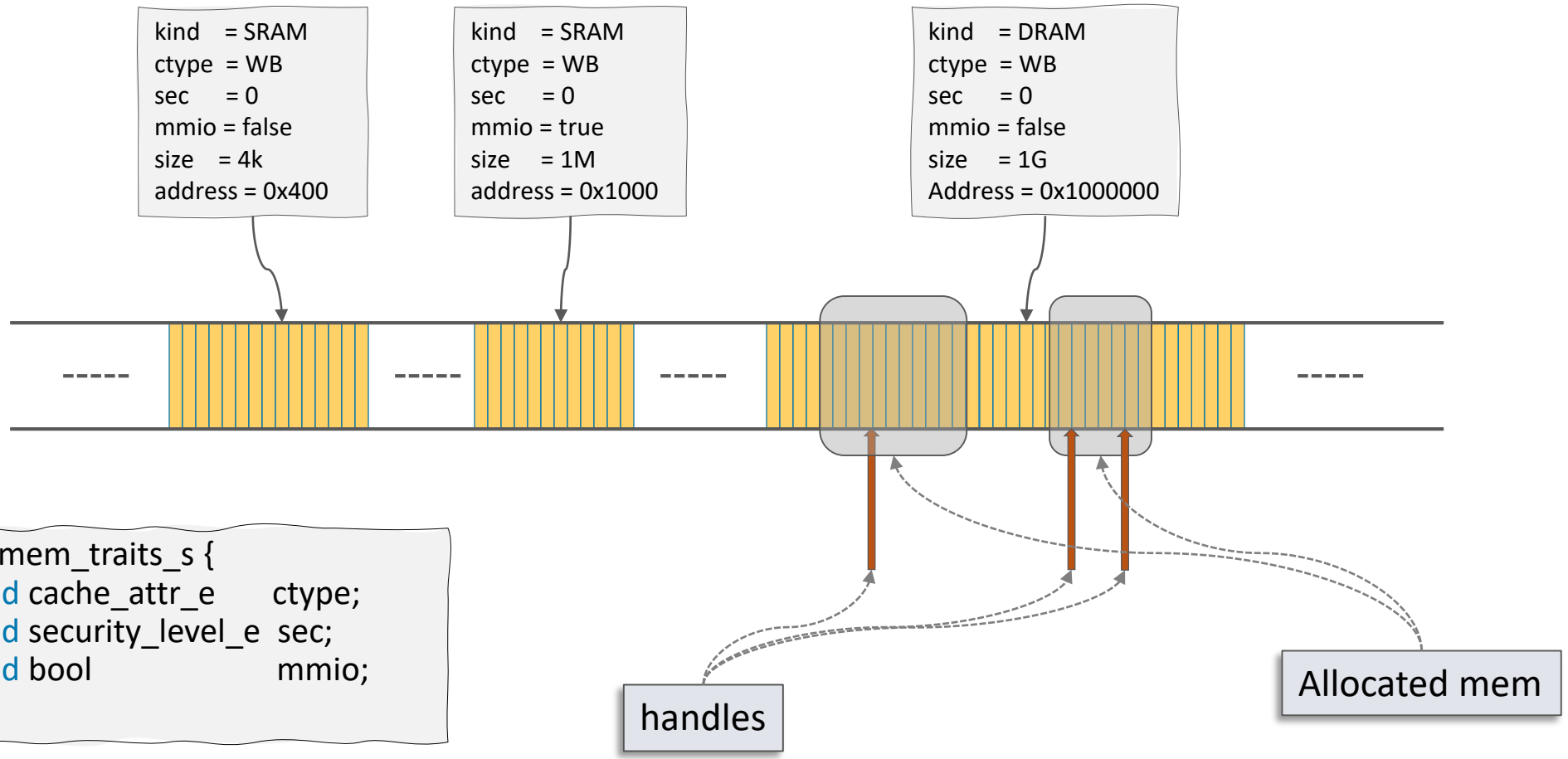
Memory claim matching

```
struct sysmem_traits_s {
    rand cache_attr_e       ctype;
    rand security_level_e  sec_level;
}
struct gfx_trait_s {}

struct sram_trait_s {}
```



1. All claims are resolved by a region in address space with trait matching allocation claim trait
2. All claims are resolved by nearest address space instance in component hierarchy that obey rule 1.

Address space handle




```
struct mem_traits_s {
    rand cache_attr_e      ctype;
    rand security_level_e  sec;
    rand bool              mmio;
}
```

Address space handle

An address handle is an opaque representation of a location in an address space. Primitive access functions use address handles for reading from and writing to an address space

PSS core library functions are used to create a handle from an address space allocation claim

PSS defines core library primitive functions that operate on a handle for reading from and writing to a *contiguous byte addressable* space

Handles can be declared in objects and actions

Memory handle example

addr_handle_t

- PSS core library type for address space handle

make_handle_from_claim

- Obtain handle from a claim in opaque manner

addr_value

- Obtain resolve address from handle only in exec body

```
action mem2mem_xfer {
   rand addr_space_pkg::addr_claim_s<mem_trait_s>  src_mem;
   rand addr_space_pkg::addr_claim_s<mem_trait_s>  dst_mem;

   constraint src_mem.size == dst_mem.size;
   rand int in [1..100] size;
   constraint size == src_mem.size;

   addr_handle_t  src_handle, dst_handle;

   exec body {

      src_handle = make_handle_from_claim(src_mem);
      dst_handle = make_handle_from_claim(dst_mem);

      bit[64] src_addr = addr_value(src_handle) >> 2;
      bit[64] dst_addr = addr_value(dst_handle) >> 2;

      // program channel registers
      // polls until status bit for dma done is 1b'1
      while (<read done bit of register>  == 0) {}
   };
};
```

Address space handle

Address handle as part of input/output buffer doesn't allocate new space

Allocation lifetime is extended by use of handle with input and output or some other persistent object

```
buffer data_buff {
  addr_handle_t mem_seg;
};
component dma_c {
  action chained_xfer {
    input  data_buff    src_buff;
    output data_buff    dst_buff;

    addr_claim_s<> claim;
    constraint claim.size == 1024;

    exec post_solve {
        dst_buff.mem_seg = make_handle_from_claim(claim);
    }
    exec body {
       int offset = 16; int data = 128;
       addr_handle_t h = make_handle_with_claim(claim, offset);

       write32(h, data);

       h = make_handle_from_handle(h, sizeof(data));

       write32(h, data);
```

Operations over byte addressable space

```
bit[8]   read8(addr_handle_t);
bit[16] read16(addr_handle_t);
bit[32] read32(addr_handle_t);
bit[64] read64(addr_handle_t);

 write8(addr_handle_t, bit[8]);
write16(addr_handle_t, bit[16]);
write32(addr_handle_t, bit[32]);
write64(addr_handle_t, bit[64]);

 read_array(addr_handle_t, list<bit[8]>);
write_array(addr_handle_t, list<bit[8]>);

 read_struct(addr_handle_t, struct);
write_struct(addr_handle_t, struct);

 read_struct_array(addr_handle_t, list<struct>);
write_struct_array(addr_handle_t, list<struct>);
```

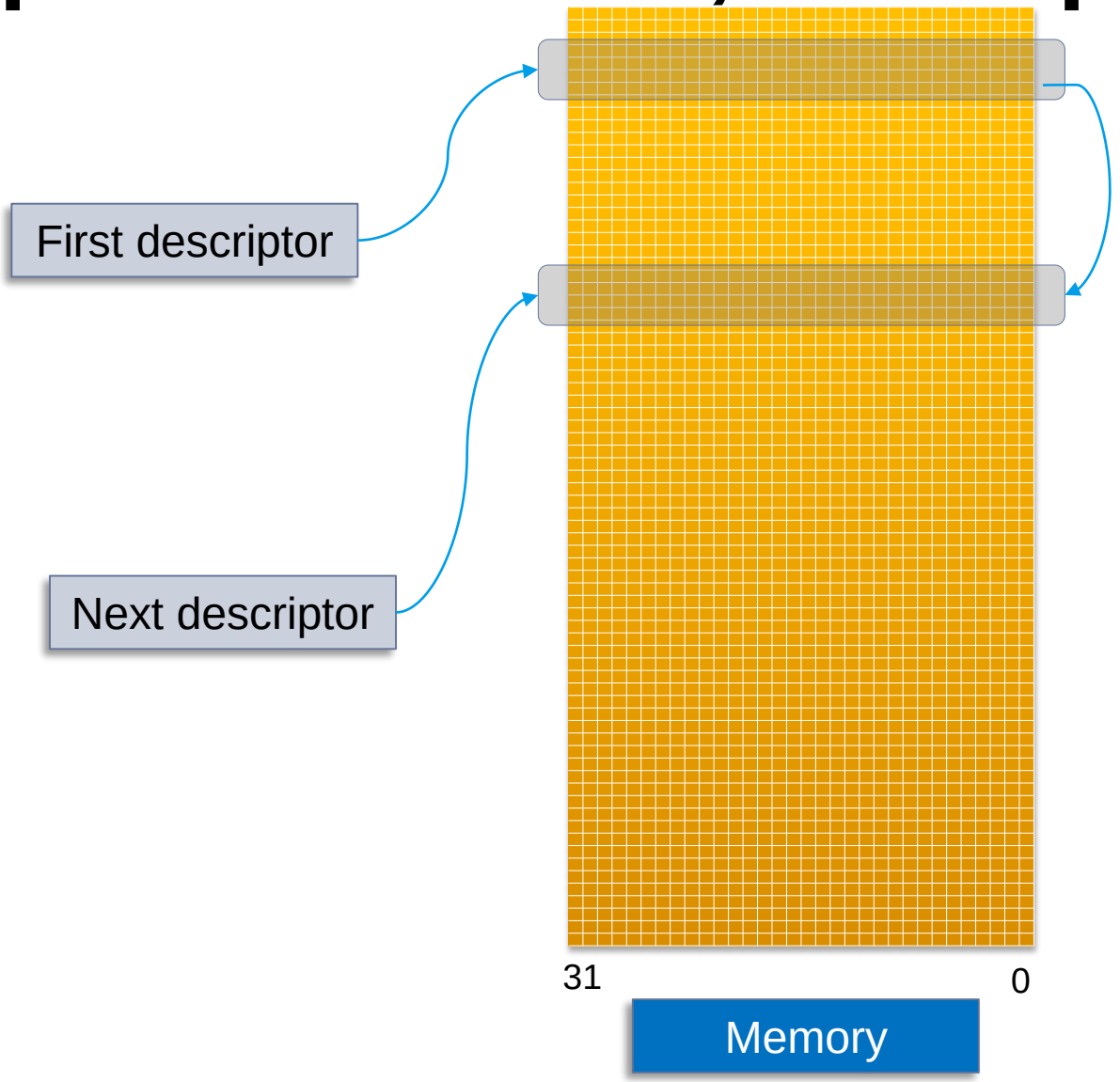
PSS defines primitive operations over byte addressable contiguous address space

Primitive operations can be customized with realization traits. PSS doesn't assign any meaning to realization traits

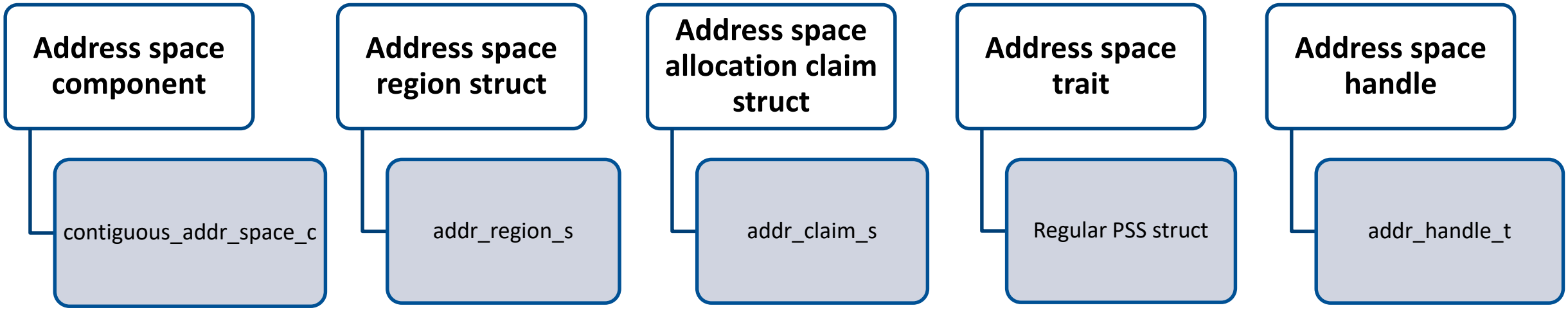
Sized handle, packed structs, descriptor

```
struct descriptor_s : packed<> {

    sized_addr_handle_s<32>  src_addr;
    sized_addr_handle_s<32>  dst_addr;
    sized_addr_handle_s<32>  next_descr;

    bit[4]    status;
    bit[12]   reserved;
    bit[16]   size;
};
```

```
// sized address handle
 struct sized_addr_handle_s
   < int SZ,
     int lsb = 0,
     endianess_e e = LITTLE_ENDIAN
   > : packed<e>
 {
       addr_handle_t hndl;
 };
```



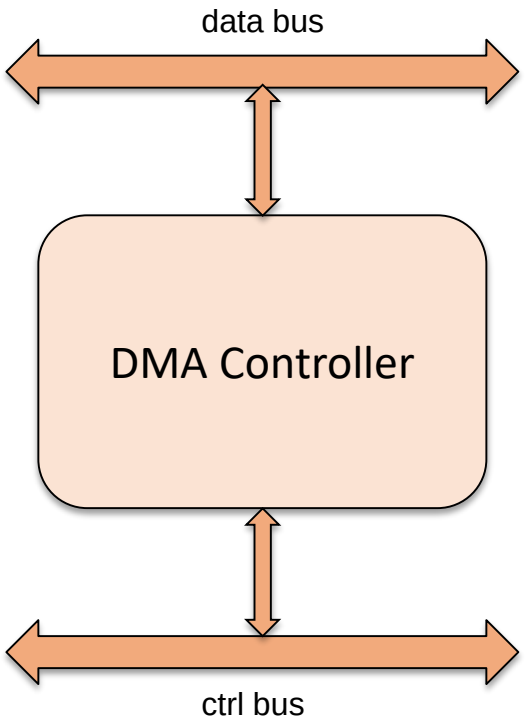
Address space APIs



DMA IP – PSS 1.0

Test space

DMA channels
DMA operation
Modes
DMA Size




```
buffer data_buff {
    rand bit[32] addr;
};

component dma_c {

   resource channel_r {};

   action mem2mem_xfer {
      input  data_buff src_buff;
      output data_buff dst_buff;

      rand int in [1..100] size;
      lock channel_r chan;
   };

   pool [NUM_DMA_CHANNELS] channel_r chan_pool;
   bind chan_pool *;
};
```

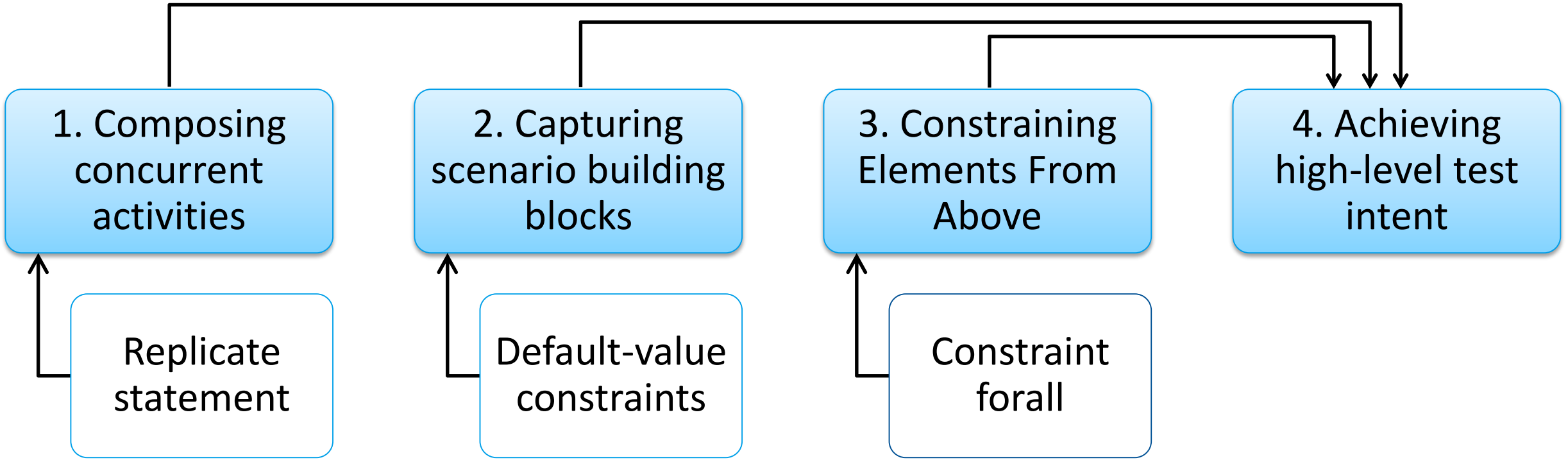
Memory allocation DMA IP PSS 1.1

- Memory allocated through flow object and actions
- Every allocation is unique
- Allocation intent portable
- Claims and handles are opaque
- Lifetime of claim can be extended with handles

```
buffer data_buff {
  rand addr_space_pkg::addr_claim_s<mem_trait_s> mem_seg;
};

component dma_c {
   action mem2mem_xfer {
      input       data_buff    src_buff;
      output      data_buff    dst_buff;
      rand int in [1..100] size;
      constraint  dst_buff.mem_seg.size == src_buff.mem_seg.size;
      addr_handle_t  src_handle, dst_handle;

      exec post_solve {
         src_handle = make_handle_from_claim(src_buff.mem_seg);
         dst_handle = make_handle_from_claim(dst_buff.mem_seg);
      };
      exec body {
         bit[64] src_addr = addr_value(src_handle) >> 2;
         bit[64] dst_addr = addr_value(dst_handle) >> 2;

         // program channel registers
         // polls until status bit for dma done is 1b'1
         while (<read done bit of register>  == 0) {}
      };
   };
};
```

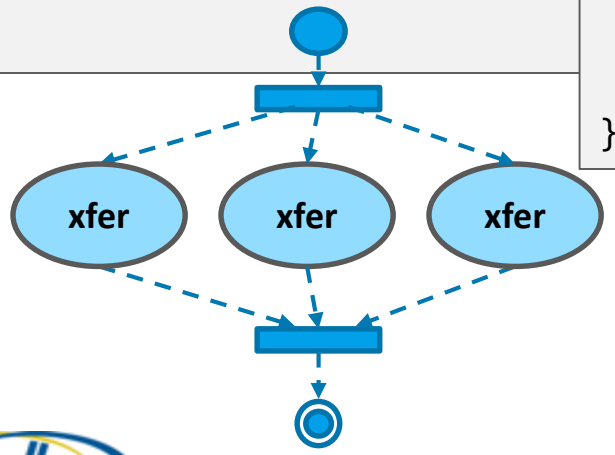
HIGH-LEVEL SCENARIOS

Stronger Scenario Language in PSS 1.1



Step 1: Composing Concurrent Activities

```
action three_parallel_xfers {
  activity {
    parallel {
      do mem2mem_xfer;
      do mem2mem_xfer;
      do mem2mem_xfer;
    }
  }
};
```




```
action all_channels_parallel_xfers {
  activity {
    parallel {
      do mem2mem_xfer;
      do mem2mem_xfer;
      do mem2mem_xfer;
      do mem2mem_xfer;
      do mem2mem_xfer;
      ...
    }
  }
};
```

X 31

What if you wanted to utilize all channels concurrently?

```
action all_channels_parallel_xfers {
  activity {
    parallel {
      replicate (NUM_CHANNELS) {
        do mem2mem_xfer;
      }
    }
  }
};
```

You really just want to replicate an activity block…

‘replicate’ statement

```
action even_channels_p
  activity {
    parallel {
      replicate (i: NUM_CHANNELS) {
        if (i%2 == 0) {
          do mem2mem_xfer with {
            channel.instance_id == i;
          }
        }
      }
    }
  }
};
```

Can also use an index to distinguish between them

replicate statement

- *Syntax*: **replicate (** [*index*_id **:**] expression **)** [*label*_id **[ ] :**] activity_stmt
- *Semantics*: In-place expansion of a specified statement multiple times

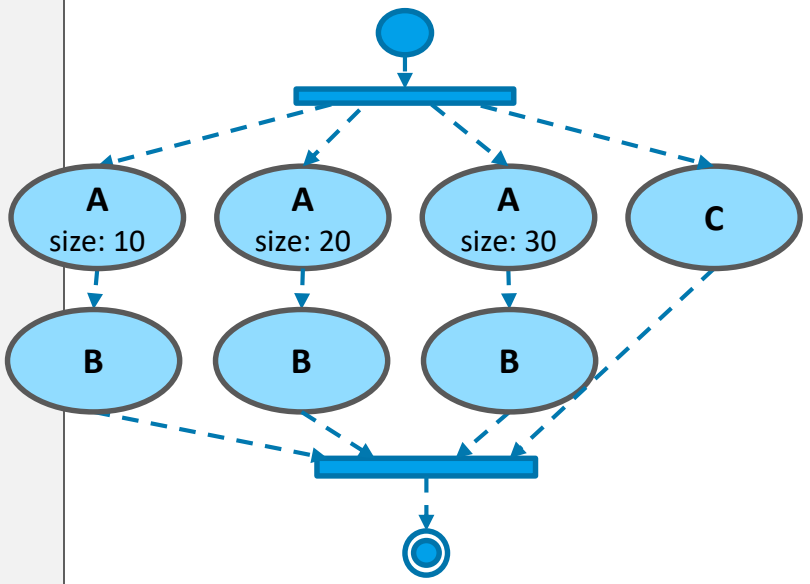
```
action my_test {
  activity {
    parallel {
      replicate (i: 3)
      {
        do A with { size == i * 10; };
        do B;
      }
      do C;
    }
  }
};
```

```
action my_test {
  activity {
    parallel {
      {
        do A with { size == 1 * 10; };
        do B;
      }
      {
        do A with { size == 2 * 10; };
        do B;
      }
      {
        do A with { size == 3 * 10; };
        do B;
      }
      do C;
    }
  }
};
```

i = 1

i = 2

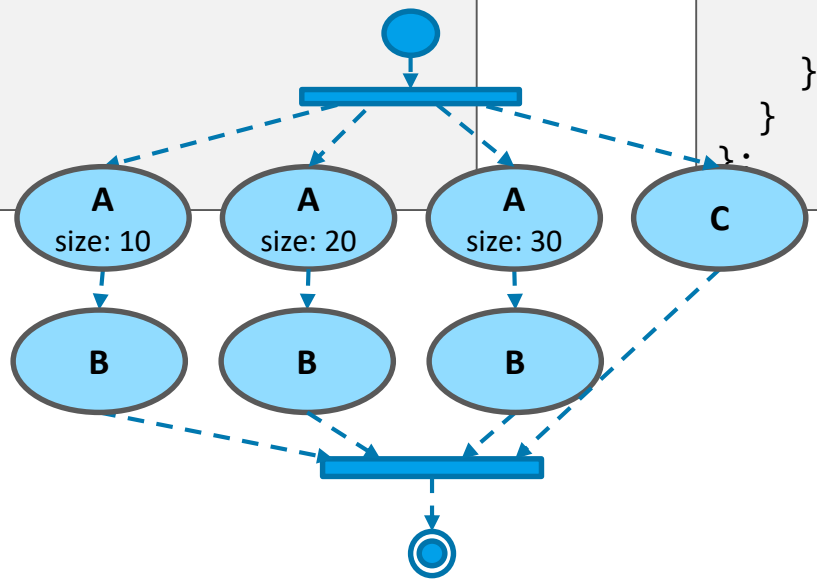
i = 3



replicate vs. *repeat*

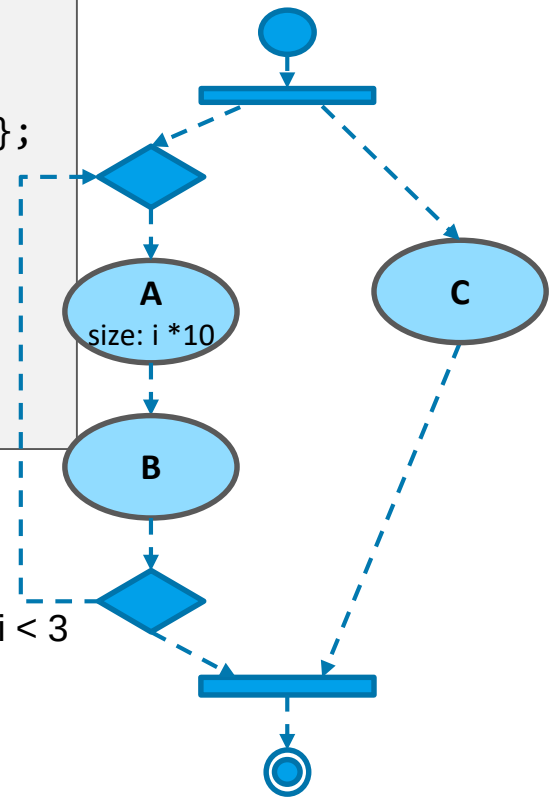
Not a loop – iterative expansion of statements

```
action my_test {
  activity {
    parallel {
      replicate (i: 3)
      {
        do A with { size == i * 10; };
        do B;
      }
      do C;
    }
  }
};
```



A *loop* – iterations executed sequentially

```
action my_test {
  activity {
    parallel {
      repeat (i: 3)
      {
        do A with { size == i * 10; };
        do B;
      }
      do C;
    }
  }
};
```



Step 2: Capturing Scenario Building Blocks

```
action parallel_mem_access {
  rand int in [2..20] num_parallel;

  activity {
    parallel {
      replicate (num_parallel) {
        select {
          do dma_c::mem2mem_xfer;
          do cpu_c::mem_copy;
        }
      }
    }
  }
};
```

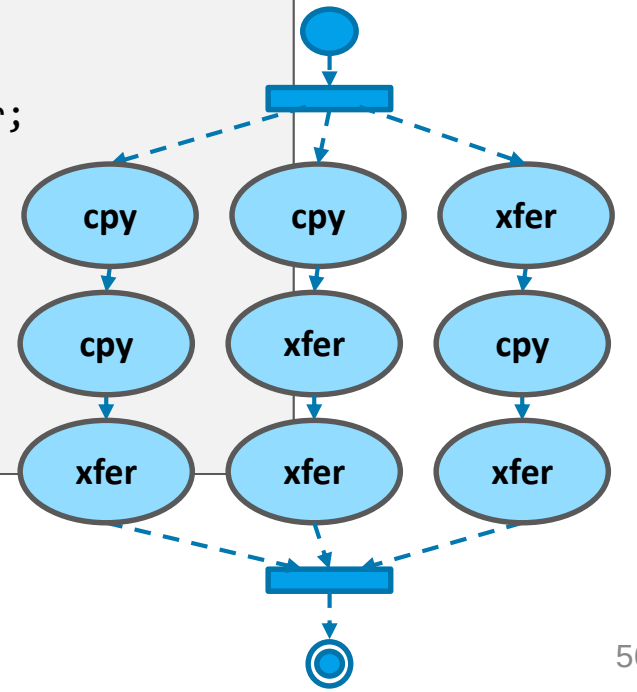
Random selection of different mem-access operations

```
action parallel_mem_access {
  rand int in [2..20] num_parallel;
  rand int in [1..100] loop_count;
  constraint default loop_count == 1;

  activity {
    parallel {
      replicate (num_parallel) {
        repeat (loop_count) {
          select {
            do dma_c::mem2mem_xfer;
            do cpu_c::mem_copy;
          }
        }
      }
    }
  }
};
```

Generalize to wider range of applications, with more control knobs

But still keep user view simple for simple cases…

default-value constraint



Default-value constraints

- *Syntax*:

  default hierarchical_id **==** constant_expression **;**

  default disable hierarchical_id ;

- *Semantics*: Determine the value of an attribute, unless explicitly disabled

```
struct my_struct {
  rand int in [0..3] attr1;
  constraint default attr1 == 0;

  rand int in [0..3] attr2;
  constraint attr1 < attr2;
};
```

```
action my_action {
  rand my_struct s1;

  rand my_struct s2;
  constraint default s2.attr1 == 2;

  rand my_struct s3;
  constraint default disable s3.attr1;
  constraint s3.attr1 > 0;
};
```

s1.attr1 is resolved to 0
s1.attr2 is randomized in the domain 1..3

s2.attr1 is resolved to 2
s2.attr2 is resolved to 3

s3.attr1 is randomized in the domain 0..2
s3.attr2 is randomized in the domain 1..3
such that its value is greater than s3.attr2

Step 3: Constraining Elements From Above

```
action three_parallel_xfers {
  dma_c::mem2mem_xfer xfer1,
                      xfer2,
                      xfer3;

  activity {
    parallel {
      xfer1;
      xfer2;
      xfer3;
    }
  }
};
```

Intent involves an additional property of all transfers

```
action large_three_parallel_xfers {
  activity {
    do three_parallel_xfers with {
      xfer1.size >= 1024;
      xfer2.size >= 1024;
      xfer3.size >= 1024;
    };
  }
};
```

Utterly impractical for complex, deep, generic activities!

```
action large_parallel_mem_access {
  activity {
    do parallel_mem_access with {
      forall (xfer: dma_c::mem2mem_xfer) {
        xfer.size >= 1024;
      }
      forall (xfer: cpu_c::mem_cpy) {
        xfer.size >= 512;
      }
    };
  }
};
```

Propagate a constraint down a hierarchy by type

constraint *forall*

Constraint *forall*

- *Syntax*:

  forall (identifier **:** type_identifier [**in** variable_ref_path] **)** constraint_set

- *Semantics*: Apply constraints to all instances of a specific type within an action / attribute subtree

```
action A {
  rand int in [0..10] v;
};

action B {
  A a1, a2;
  activity {
    parallel { a1; a2; }
  }
};
```

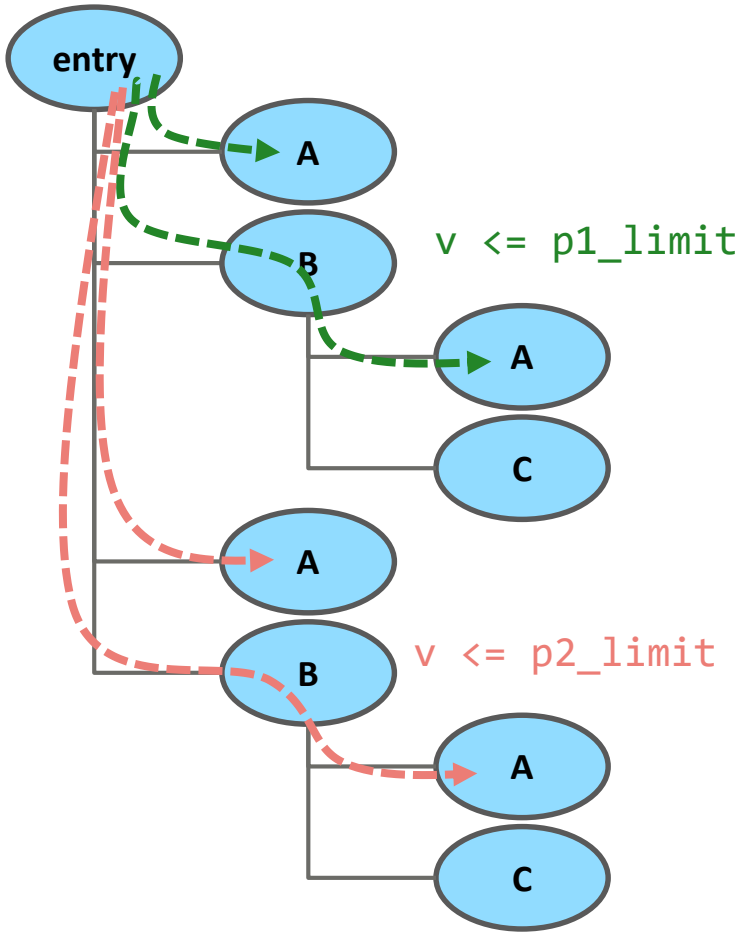
```
action entry {
  rand int in [0..10] limit;
  A a;
  B b;
  constraint forall (a_it: A) {
    a_it.a <= limit;
  }
  activity {
    schedule { a; b; }
  }
};
```

```
action entry {
  rand int v_limit;
  A a;
  B b;
  constraint {
    a.v    <= limit;
    b.a1.v <= limit;
    b.a2.v <= limit;
  }
  activity {
    schedule { a; b; }
  }
};
```

Constraint *forall* Scoping

```
action A {
  rand int in [0..10] v;
};

action B {};

action C {
  A a;
  B b;
  activity {
    parallel { a; b; }
  }
};
```

```
action entry {
  rand int in [0..10] p1_limit;
  rand int in [0..10] p2_limit;

  activity {
    schedule {
      do A; do B;
      constraint forall (a_it: A) {
        a_it.v <= p1_limit; }
    }
    schedule {
      do A; do B;
      constraint forall (a_it: A) {
        a_it.v <= p2_limit; }
    }
  }
};
```

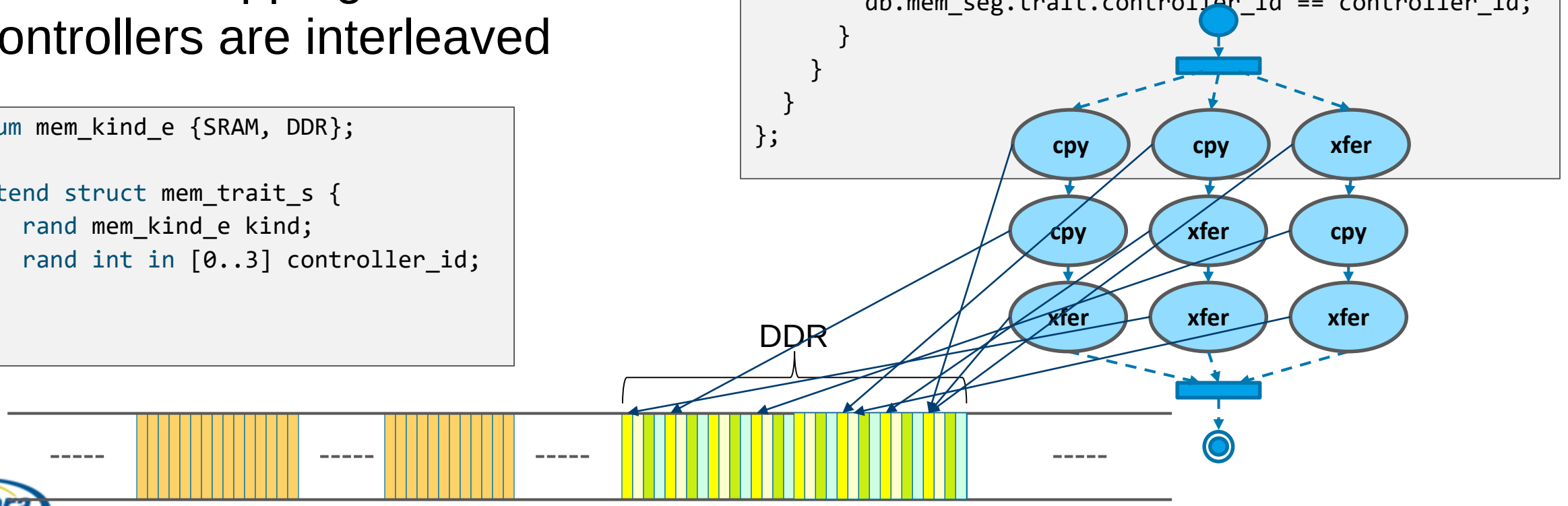


Step 4: Achieving High-level Test Intent…

- Access same DDR controller from many agents concurrently
- Address mapping of DDR controllers are interleaved

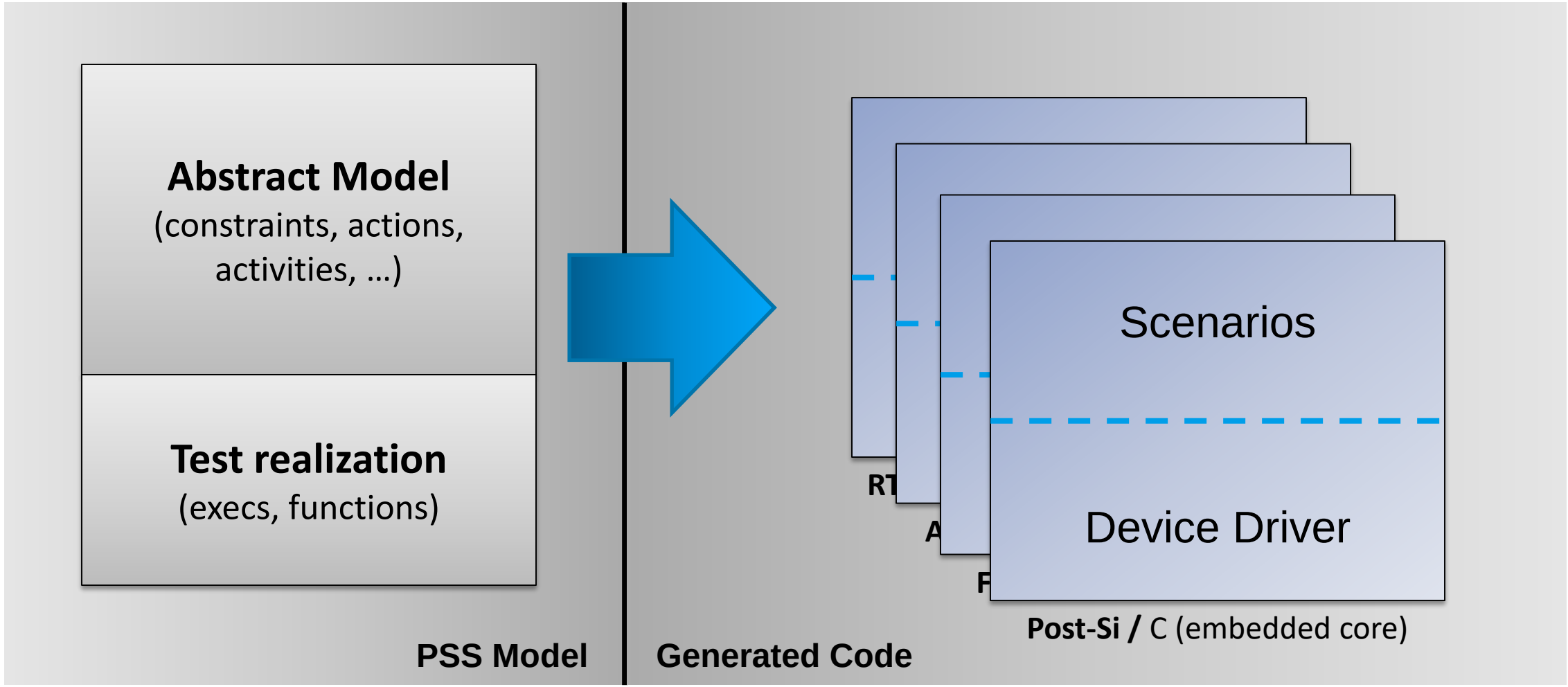
```
action ddr_stress {
  rand int in [0..3] controller_id;
  activity {
    do parallel_mem_access with {
      forall (db: data_buff) {
        db.mem_seg.trait.kind == DDR;
        db.mem_seg.trait.controller_id == controller_id;
      }
    }
  }
};
```

```
enum mem_kind_e {SRAM, DDR};

extend struct mem_trait_s {
    rand mem_kind_e kind;
    rand int in [0..3] controller_id;
};
```

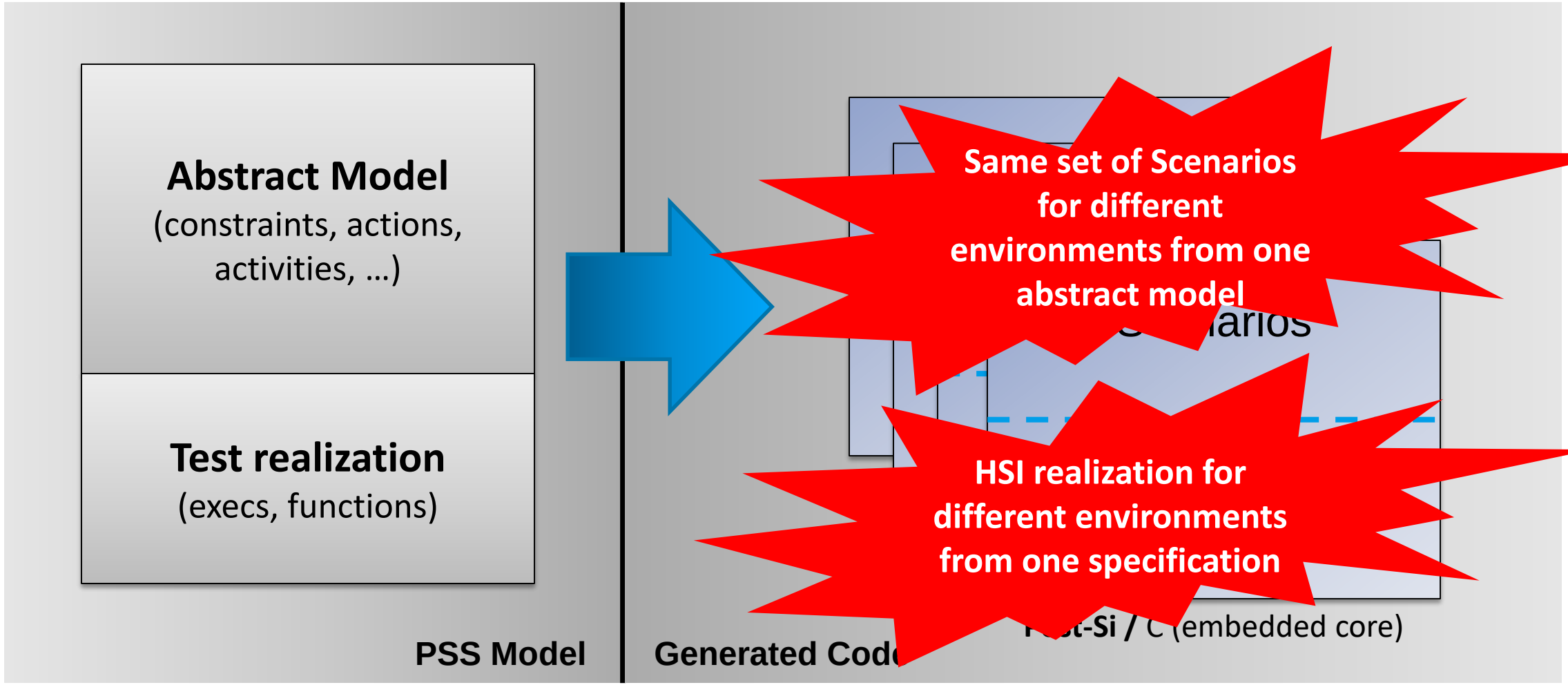


HSI REALIZATION

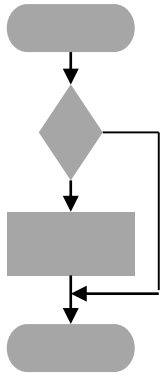
HSI Realization: Motivation



HSI Realization: Motivation

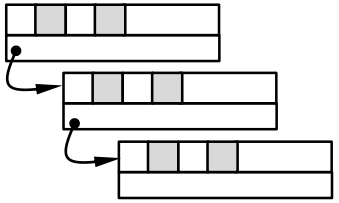


Relevant Enhancements in PSS1.1



Procedural constructs

Programmable Registers



Descriptors

Sweet spot: Capture device programming sequence

Procedural Constructs Introduction

```
function int glob_func();
import solve function glob_func();

component my_comp_c
{
  function void comp_func();
  target ASM function void comp_func() = """ ... """;

  action act_a {
    exec pre_solve {
      // Only assignments, function calls and "super"
    }
    exec body {
      // Only assignments, function calls and "super"
    }
  };
};
```

my_model.pss

- PSS1.0
  - Foreign-language functions can be imported or specified with target-templates
  - Exec definitions are restricted

Procedural Constructs Introduction

```
function int glob_func() {
  // Define glob_func
}
component my_comp_c
{
  function void comp_func() {
    // Define comp_func
  }
  action act_a {
    exec pre_solve {
      // Expanded set of supported statements
    }
    exec body {
      // Expanded set of supported statements
    }
  };
};
```

my_model.pss

- PSS1.0
  - Foreign-language functions can be imported or specified with target-templates
  - Exec definitions are restricted
- PSS1.1 adds,
  - Support for a generic function definition
  - Many more procedural statements
    - Can be used in execs and function definition

Procedural Constructs Introduction

```
// Sum all elements of 'a' that are even, starting from a[0], except those
// that are equal to 42.  Stop summation if the value of an element is 0
function int sum(int a[100])
{
   int res;

   res = 0;

   foreach (el : a) {
      if (el == 0)
         break;
      if (el == 42)
         continue;
      if ((el % 2) == 0) {
         res = res + el;
      }
   }

   return res;
}
```

sum.pss

- Variables can be declared
- Conditional branches, Loops, Match statement supported
  - Similar to activity statement
- Break / Continue
- Return

- No randomization, constraints (algebraic, schedule)

Example IP: ARM PL080 DMAC

Programming a DMA channel

1. Clear any pending interrupts
2. Setup the channel registers
   - Set the source address
   - Set the destination address
   - Set the address of the next LLI
   - Write the control information
3. Enable the DMA channel
   - Write the channel configuration information

DMAC: A Component Definition

```
buffer data_buff {
  addr_handle_t mem_seg;
};

component dma_c {
  resource channel_r {};
  pool [NUM_DMA_CHANNELS] channel_r chan_pool;
  bind chan_pool *;


  action mem2mem_xfer {
    input  data_buff src_buff;
    output data_buff dst_buff;


    addr_claim_s<> claim;
    constraint claim.size == 1024;
    lock channel_r chan;

    // contd...
```

```
    exec post_solve {
      dst_buff.mem_seg =
         make_handle_from_claim(claim);
    }

    exec body {
      comp.do_xfer(chan,
        src_buff.mem_seg,
        dst_buff.mem_seg,
        claim.size);
    }
  }; // action mem2mem_xfer

 function void do_xfer(int channel,
    addr_handle_t src,
    addr_handle_t dst,
    int length);

}; // component dma_c
```

dmac.pss

PL080: Component extensions

```
// Recommended: Extend or derive the component
// while using design-specific registers
extend component dma_c
{
  PL080_regs::Regs_c r;

  function void do_xfer(int channel,
      addr_handle_t src,
      addr_handle_t dst,
      int length) { /* Details in later slide */ }
};
```

pl080_c.pss

```
// Recommended: Keep register definitions
// in a separate file
package PL080_regs {
  struct INT_TC_CLR_s : packed<>
  {
     bit TC_CLR[8];
  };
  pure component INT_TC_CLR_c : reg_c<INT_TC_CLR_s, 32>
  { };

  // ... Etc for all registers

  pure component Regs_c : reg_group_c
  {
    INT_TC_CLR_c  INT_TC_CLR;
    INT_ERR_CLR_c INT_ERR_CLR;
    SRC_ADDR_c    SRC_ADDR[8];
    DST_ADDR_c    DST_ADDR[8];
    // ...
  };
};
```

pl080_regs.pss

PL080: Component extensions

- Every register is associated with:
  - A struct type representing the value in the register
  - A component type
    - For instantiating a register in a register group
    - Defines functions that can be invoked on the register
- Register group
  - Instantiates registers (and possibly other register groups)
  - Associates register with an offset (not shown in example)
  - A “top-level” group is associated with a handle to an address region

pl080_c.pss

pl080_regs.pss

PL080: Setup Transfer

```
function void do_xfer(int channel, addr_handle_t src, addr_handle_t dst, int length)
{
  // Clear Interrupts
  comp.r.INT_TC_CLR.write_val(0xF);
  comp.r.INT_ERR_CLR.write_val(0xF);

  // Setup channel
  comp.r.SRC_ADDR[channel].write(src);
  comp.r.DST_ADDR[channel].write(dst);
  comp.r.LLI[channel].write_val(0);
  comp.r.CONTROL[channel].write_val(length);

  // Enable channel
  CONFIGURATION_s cfg;
  cfg.Enable = 1;
  comp.r.CONFIGURATION[channel].write(cfg);

  // Wait for completion
  INT_TC_STATUS_s sts;
  repeat {
    yield();
    sts = comp.r.INT_TC_STATUS.read();
  } while(sts.TC_STS[channel] == 0);
}
```


pl080_c.pss

PL080: Setup Transfer

```
function void do_xfer(int channel, addr_handle_t src, addr_handle_t dst, int length)
{
  // Clear Interrupts
  comp.r.INT_TC_CLR.write_val(0xF);
  comp.r.INT_ERR_CLR.write_val(0xF);

  CONFIGURATION_s cfg;
  cfg.Enable = 1;
  comp.r.CONFIGURATION[channel].write(cfg);

  // Wait for completion
  INT_TC_STATUS_s sts;
  repeat {
    yield();
    sts = comp.r.INT_TC_STATUS.read();
  } while(sts.TC_STS[channel] == 0);
}
```

Instance of

```
struct INT_TC_CLR_s : packed<>
{
   bit TC_CLR[8];
};
pure component INT_TC_CLR_c : reg_c<INT_TC_CLR_s, 32> {};
```

pl080_c.pss

PL080: Setup Transfer

```
function void do_xfer(int channel, a
{
  // Clear Interrupts
  comp.r.INT_TC_CLR.write_val(0xF);
  comp.r.INT_ERR_CLR.write_val(0xF);

  CONFIGURATION_s cfg;
  cfg.Enable = 1;
  comp.r.CONFIGURATION[channel].writ

  // Wait for completion
  INT_TC_STATUS_s sts;
  repeat {
    yield();
    sts = comp.r.INT_TC_STATUS.read(
  } while(sts.TC_STS[channel] == 0);
}
```

```
struct INT_TC_CLR_s : packed<>
{
   bit TC_CLR[8];
};
pure component INT_TC_CLR_c : reg_c<I
```

```
enum reg_access { READWRITE, READONLY, WRITEONLY};
pure component reg_c <
  type R,
  int SZ = typeinfo<R>::size,
  reg_access ACC = READWRITE>
{
  // Read/write the register as a 'struct'
  function R read();
  import target function R read();
  function void write(R r);
  import target function void write(R r);

  // Read/write as bits
  function bit[SZ] read_val();
  import target function bit[SZ] read_val();
  function void write_val(bit[SZ] r);
  import target function void write_val(bit[SZ] r);
};
```

pl080_c.pss

PL080: Setup Transfer

```
struct CONFIGURATION_s : packed<>
{
    bit Enable;
    bit M1;
    bit M2;
};
pure component CONFIGURATION_c : reg_c<CONFIGURATION_s, 32> {};
```

Instance of

```
function void do_xfer(int channel, addr_handle_t src, addr_handle_t dst, int length)
{
  // Enable channel
  CONFIGURATION_s cfg;
  cfg.Enable = 1;
  comp.r.CONFIGURATION[channel].write(cfg);

  // Wait for completion
  INT_TC_STATUS_s sts;
  repeat {
    yield();
    sts = comp.r.INT_TC_STATUS.read();
  } while(sts.TC_STS[channel] == 0);
}
```

pl080_c.pss

PL080: Setup Transfer

```
function void do_xfer(int channel, addr
```

```
struct CONFIGURATION_s : packed<>
{
    bit Enable;
    bit M1;
    bit M2;
};
pure component CONFIGURATION_c : reg_c<CO
```

```
enum reg_access { READWRITE, READONLY, WRITEONLY};
pure component reg_c <
  type R,
  int SZ = typeinfo<R>::size,
  reg_access ACC = READWRITE>
{
  // ...
  function void write(R r);
  import target function void write(R r);
};
```

```
  // Enable channel
  CONFIGURATION_s cfg;
  cfg.Enable = 1;
  comp.r.CONFIGURATION[channel].write(cfg);

  // Wait for completion
  INT_TC_STATUS_s sts;
  repeat {
    yield();
    sts = comp.r.INT_TC_STATUS.read();
  } while(sts.TC_STS[channel] == 0);
}
```

pl080_c.pss

PL080: Setup Transfer

```
function void do_xfer(int channel, addr
{
  // Clear Interrupts
  comp.r.INT_TC_CLR.write_val(0xF);
  comp.r.INT_ERR_CLR.write_val(0xF);

  // Setup channel
  comp.r.SRC_ADDR[channel].write(src);
  comp.r.DST_ADDR[channel].write(dst);
  comp.r.LLI[channel].write_val(0);
  comp.r.CONTROL[channel].write_val(ler

  // Enable channel
  CONFIGURATION_s cfg;
  cfg.Enable = 1;
  comp.r.CONFIGURATION[channel].write(cfg);

  // Wait for completion
  INT_TC_STATUS_s sts;
  repeat {
    yield();
    sts = comp.r.INT_TC_STATUS.read();
  } while(sts.TC_STS[channel] == 0);
}
```

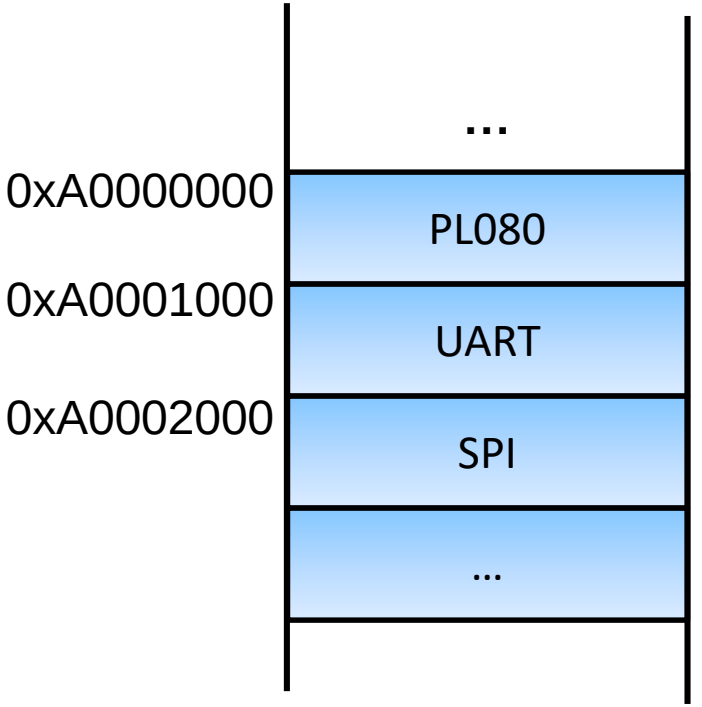
```
enum reg_access { READWRITE, READONLY, WRITEONLY};
pure component reg_c <
  type R,
  int SZ = typeinfo<R>::size,
  reg_access ACC = READWRITE>
{
  // ...
  function R read();
  import target function R read();
};
```

pl080_c.pss

PL080: Stitching it all together

```
component my_top_level_c
{
  dma_c  pl080;
  uart_c uart;
  spi_c  spi;
  // etc...

  // Represents the system address map
  transparent_addr_space_c<> sys_mem;

  exec init {
    transparent_addr_region_s<> pl080_region;
    pl080_region.size = 1024;
    pl080_region.address = 0xA0000000;

    sys_mem.add_nonallocatable_region(pl080_region);

    pl080.r.set_handle(make_handle_from_region(pl080_region));

    // Likewise for other components
  }
};
```

top_level.pss



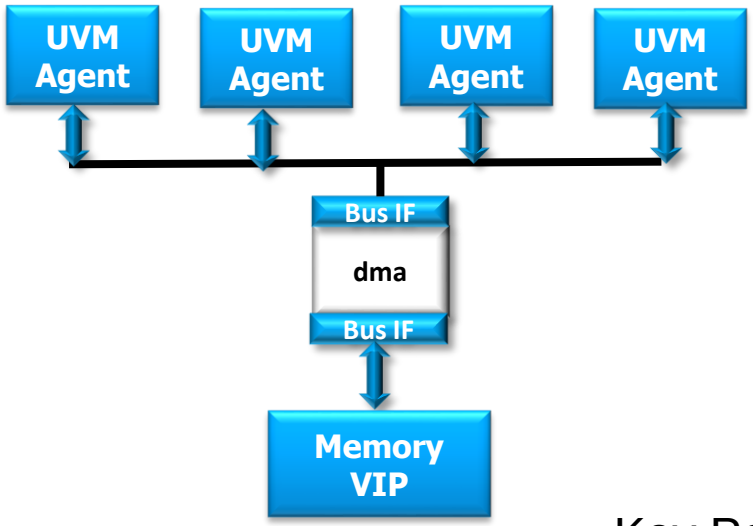
Test realization: Other considerations

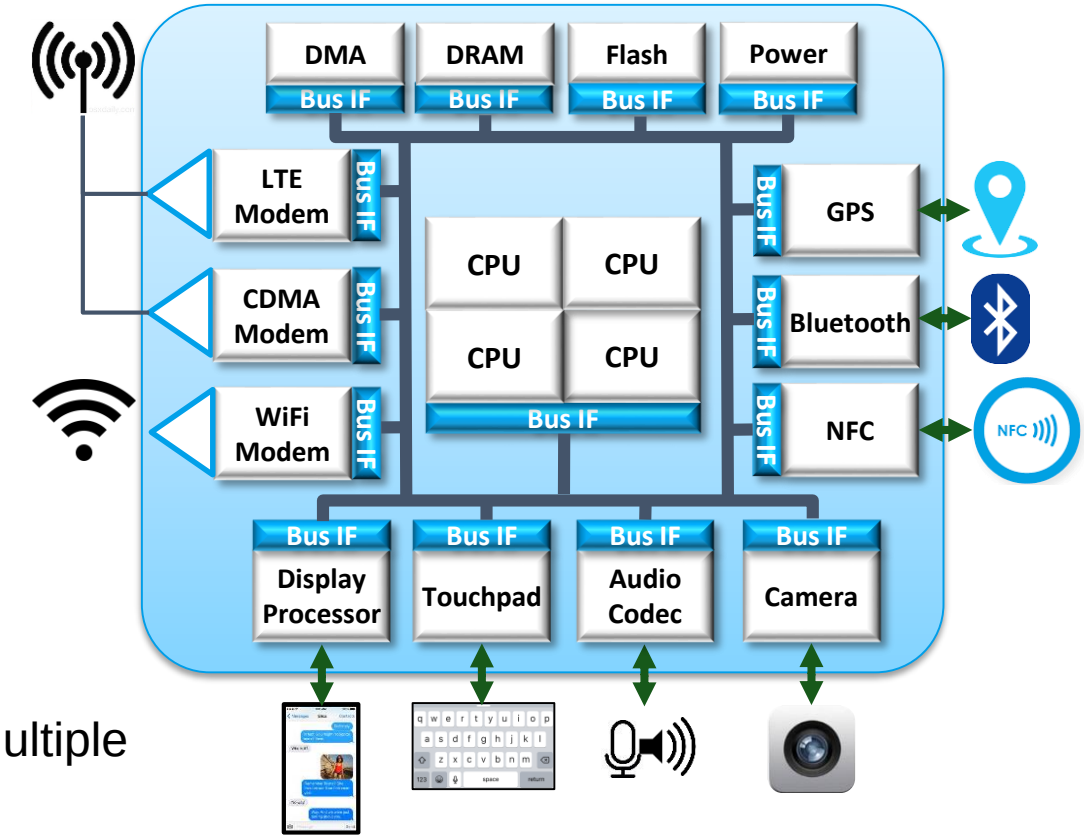
- Recommendations
  - Keep register types in a separate file – typically generated from IP-XACT/SystemRDL
  - Use registers in an extension of the component
- PSS1.1 defines constructs to enable optimized representation of large static structures (like registers)
  - Not shown in the example
- Equivalent constructs in PSS-C++ are specified

SYSTEM-LEVEL USAGE

A Block-to-System Portability and Productivity

Verification productivity goal #1: Re-Use configuration programming UVM IP Bus to SOC Bus





Block

System

Key Points for this section

- Targeting realization for multiple environments
- Reusable composition of scenarios adaptive to different integrations

IP to SOC Portability Modeling

Modeling principles

- Abstract Stimulus Model Target Platform Agnostic
  - Clean Functional Stimulus Model
- Encapsulation of target-specific implementation
  - Interchangeable to enable Portability
- Solve Time Decisions on how Test Realization is Mapped
  - Minimize Algorithmic Decisions at Runtime
- Separation of Target Platform Concerns in Different Files
  - Different Set of Files Used for Different Platforms

IP to SOC – Functional Model

Target Platform Agnostic

- address space resolved via comp tree

```
extend component dma_c {
  action configure_a {
    rand addr_space_pkg::addr_claim_s<mem_trait_s> claim;
    constraint claim.size == 4;
    addr_handle_t hndl;
    share execution_agent_r execution_agent;
    exec body {
      // Writing to memory
      hndl = make_handle_from_claim(claim);
      write32(hndl, data);
      // program channel registers
      comp.regs.ch_group[chan.instance_id].dma_ch_sz.write(sz);
      comp.regs.ch_group[chan.instance_id].dma_ch_csr.write(csr);
    }
  }
}
component pss_top {
  dma_c dma1, dma2;
  action entry {
    activity {
      do dma_c::configure_a with {comp == pss_top.dma1;}
      do dma_c::configure_a with {comp == pss_top.dma2;}
    }
  }
}
```

```
// Generated C code
(*((volatile int*) 0x10000000) = 0x00000005;
(*((volatile int*) 0x2000ff00) = 0x00000010;
(*((volatile int*) 0x2000ff04) = 0x00000015;
```

```
// Generated UVM code
write("test.env.master[0]",
  0x10000000, 0x00000005);
regs.ch_group[0].dma_ch_sz.write(status, 10);
regs.ch_group[0].dma_cdr_sz.write(status, 15);
```

dma_fm.pss

Execution Agents

```
extend component pss_top {
  resource execution_agent_r {}
  pool [4] execution_agent_r execution_agent_p;
  bind execution_agent_p *;
}
```

pss_top_execution_agent.pss

Different Target Platforms will have a different number of execution agents

Separation of concerns

Attributes to characterize behavior of specific execution agent

Address Space

```
extend component pss_top {
  contiguous_addr_space_c<mem_trait_s> mem_addr_space;
  addr_region_s<mem_trait_s> dram_region;
  addr_region_s<mem_trait_s> flash_region;
  exec init {
    dram_region.trait.kind = DRAM;
    mem_addr_space.add_region(dram_region);
    mem_addr_space.add_region(flash_region);
  }
}
```

pss_top_address_space.pss

UVM Separation of Concerns

Separation of Target Platform Concerns in Different Files

```
package my_uvm_pkg {
  // Execution agent resource instance_id can be mapped
  // to entry in list to provide tool specific
  // information.
  const list<string> execution_agent_map = {
      "test.env.master[0]", "test.env.master[1]",
      "test.env.master[2]", "test.env.master[3]"
    };
  }
}
```

my_uvm_pkg.pss

```
extend component pss_top {
  import my_uvm_pkg::*;
}
```

pss_top_uvm.pss

For illustrative purposes
There are other valid approaches to define target-specific information to a testbench generator

EMBC Separation of Concerns

Separation of Target Platform Concerns in Different Files

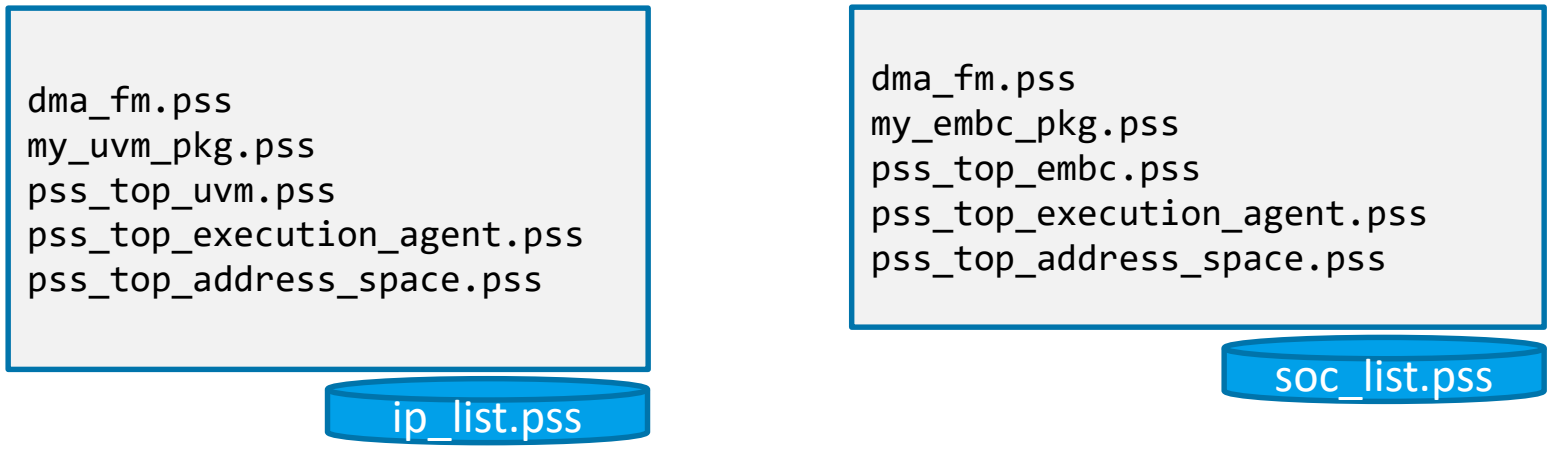
```
package my_embc_pkg {
  const list<string> execution_agent_map = {
    “cluster:0,core:0”, “cluster:0,core:1”,
    “cluster:1,core:0”, “cluster:1,core:1”
  };
}
```

my_embc_pkg.pss

```
extend component pss_top {
  import my_embc_pkg::*;
}
```

pss_top_embc.pss

File Lists



dma_fm.pss is the platform-independent abstract model
other files define target-specific information

Mixed Target Platform Package

Separation of Target Platform Concerns in Different Files

```
package my_mixed_pkg {
  const list<string> execution_agent_map = {
    "test.env.master[0]", "test.env.master[1]“,
    “cluster:1,core:0”, “cluster:1,core:1”
  };
}
```

my_uvm_pkg.pss

```
extend component pss_top {
  import my_mixed_pkg::*;
}
```

pss_uvm_top.pss

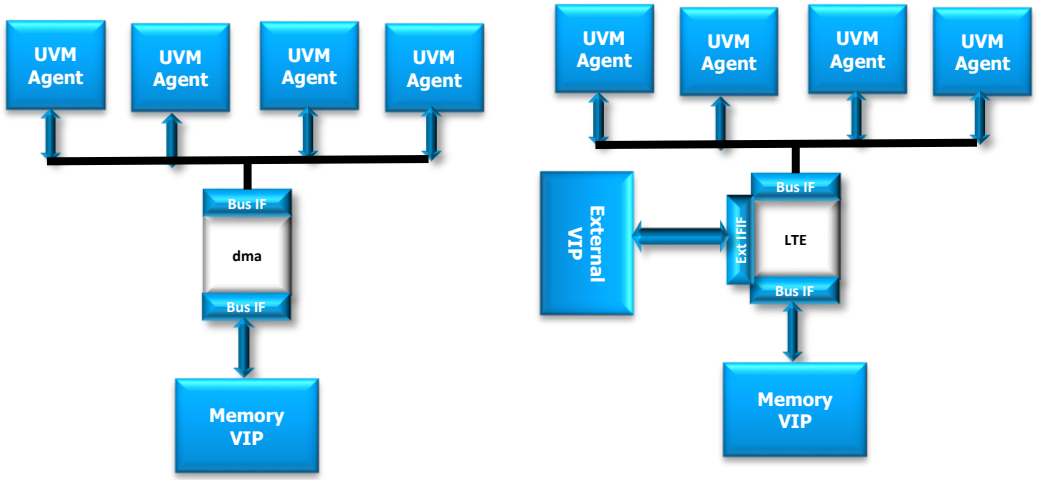
Mixed Targeting

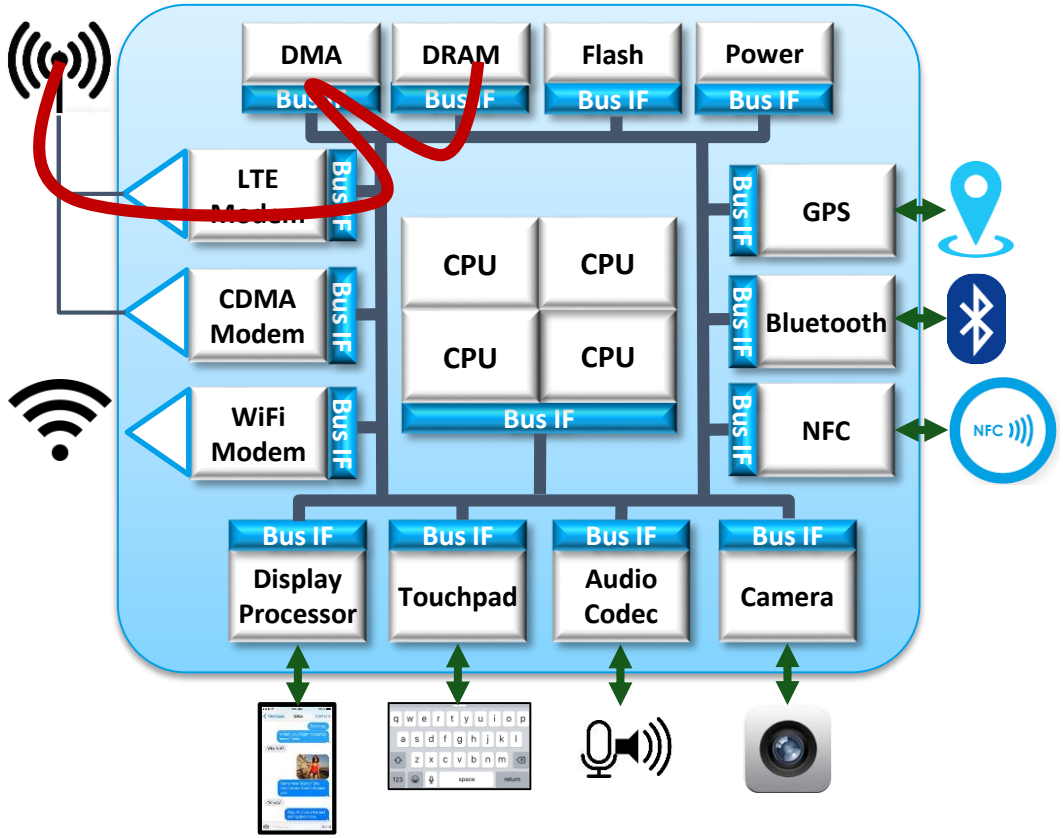
```
extend component dma_c {
  action configure_a {
    rand addr_space_pkg::addr_claim_s<mem_trait_s> claim;
    constraint claim.size == 4;
    addr_handle_t hndl;
    share execution_agent_r execution_agent;
    exec body {
      // Writing to memory
      hndl = make_handle_from_claim(claim);
      write32(hndl, data);
      // program channel registers
     comp.regs.ch_group[chan.instance_id].dma_ch_sz.write(sz);
     comp.regs.ch_group[chan.instance_id].dma_ch_csr.write(csr);
    }
  }
}
component pss_top {
  dma_c dma1, dma2;
  action entry {
    activity {
      do dma_c::configure_a with {comp == pss_top.dma1;}
      do dma_c::configure_a with {comp == pss_top.dma2;}
    }
  }
}
```

```
extend component pss_top {
  extend action entry {
    constraint forall (it_a:dma_c::*) {
      if (it_a.comp == pss_top.dma1)
        execution_agent.instance_id in [0..1]
      else
        execution_agent.instance_id in [2..3]
    }
  }
}
```

A Block-to-System Portability and Productivity

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





Block

System

System Level Definition

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

  // Execution agents
  pool [4] cpu_r cpu;
  pool [1] lte_vip_r lte_vip;
  …
}
```

```
extend component pss_top {
  // Address Space
  contiguous_addr_space_c<mem_trait_s> mem_addr_space;
  addr_region_s<mem_trait_s> dram_region;
  addr_region_s<mem_trait_s> flash_region;
  exec init {
    dram_region.trait.kind = DRAM;
    mem_addr_space.add_region(dram_region);
    mem_addr_space.add_region(flash_region);
  }
}
```

Chaining Stimulus from Multiple IP through Memory Buffers

Common Flow Object Type to Declare Output Buffer

Chaining Structures: Sequential, Parallel, Graphs

Usage of Storage Allocation for Data Integrity

Chaining Coverage

Common Flow Object Type to Declare Output Buffer

```
package common_target {
  buffer common_target_buffer {
    rand addr_space_pkg::addr_claim_s<mem_trait_s> mem_seg;
  }

  abstract action mem_copy_a {
    input common_target_buffer ctb_in;
    output common_target_buffer ctb_out;
  }
}
```

IP Owners Stimulus

LTE Modem

CDMA Modem

WiFi Modem

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

GPS

Bluetooth

NFC

Sink:
Memory to External VIP

Audio Codec

Display Processor

Source:
External VIP to Memory

Touchpad

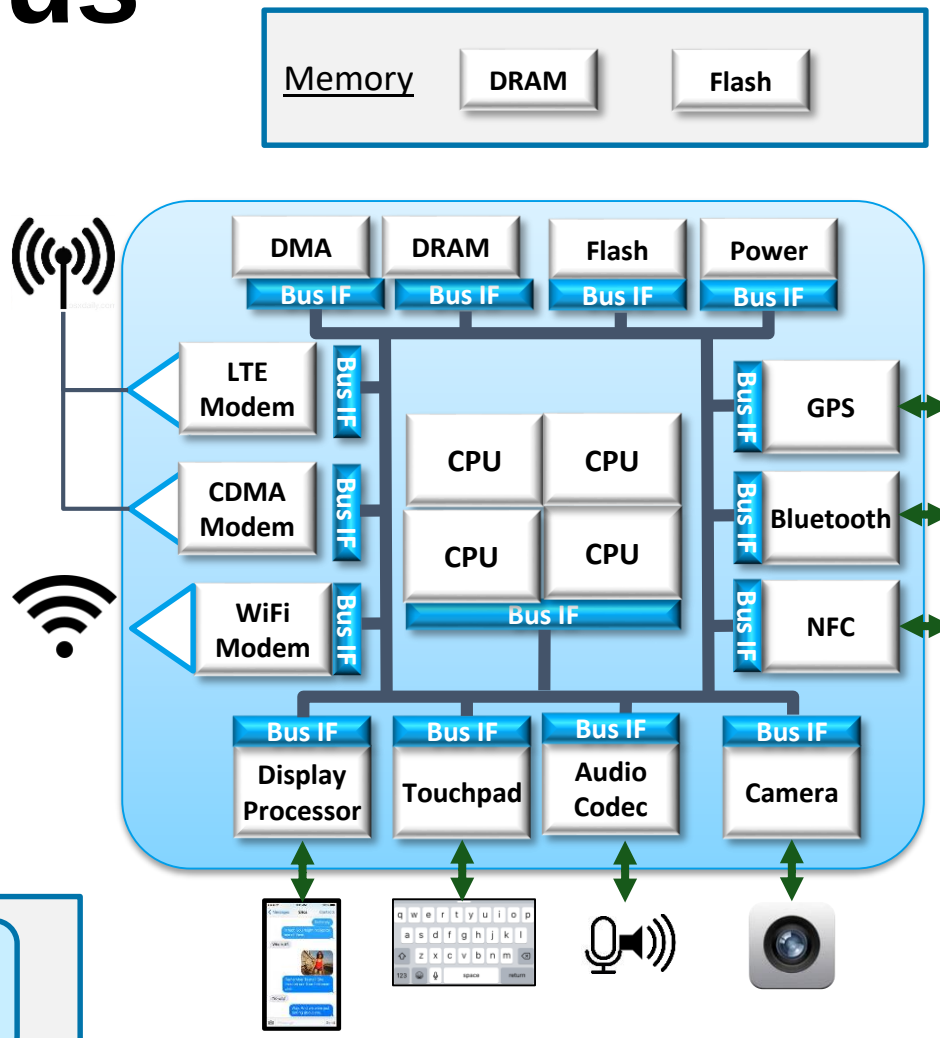
Camera

DMA

Memory2Memory:
DMA Transfer

CPU

Memory2Memory:
(a) Read Memory
(b) Write Memory



IP Owners Action

LTE Modem

CDMA Modem

WiFi Modem

```
action <device-name>_source_a {
  output common_target_buffer ctb_out;
  …
}
action <device-name>_mem2mem_a :
  mem_copy_a {…}
```

GPS

Bluetooth

NFC

```
action_<device-name>_sink_a {
  input common_target_buffer ctb_in;
  …
}
```

Audio Codec

Display Processor

```
action <device-name>_source_a {
output common_target_buffer ctb_out;
  …
}
```

Touchpad

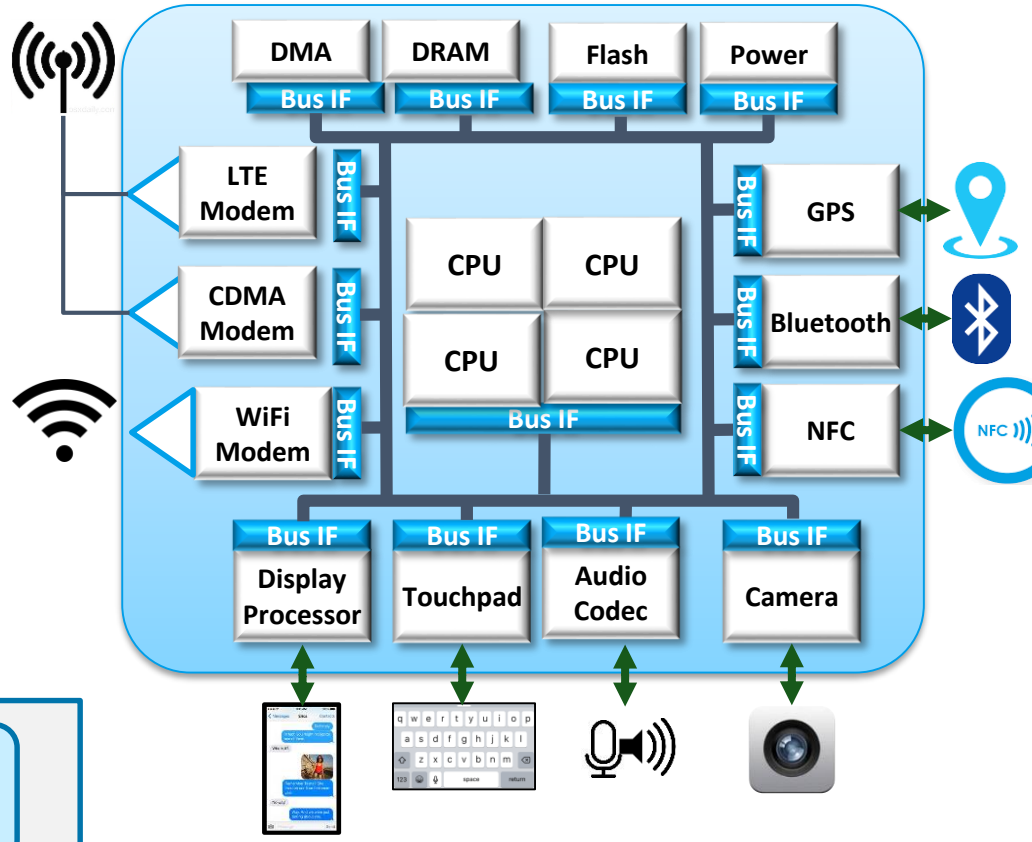
Camera

DMA

```
action core_mem2mem_a :
  mem_copy_a {…}
```

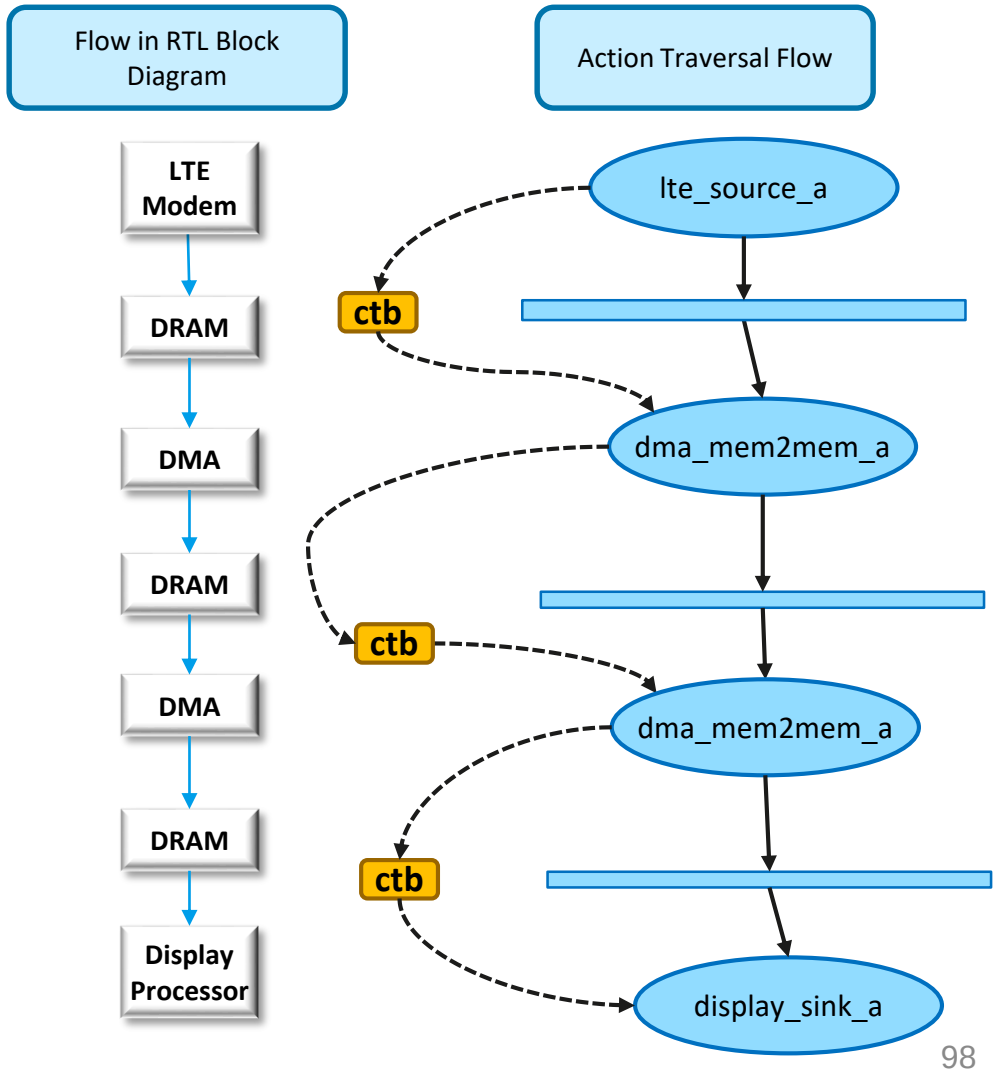
CPU

```
action core_mem2mem_a :
  mem_copy_a {…}
```



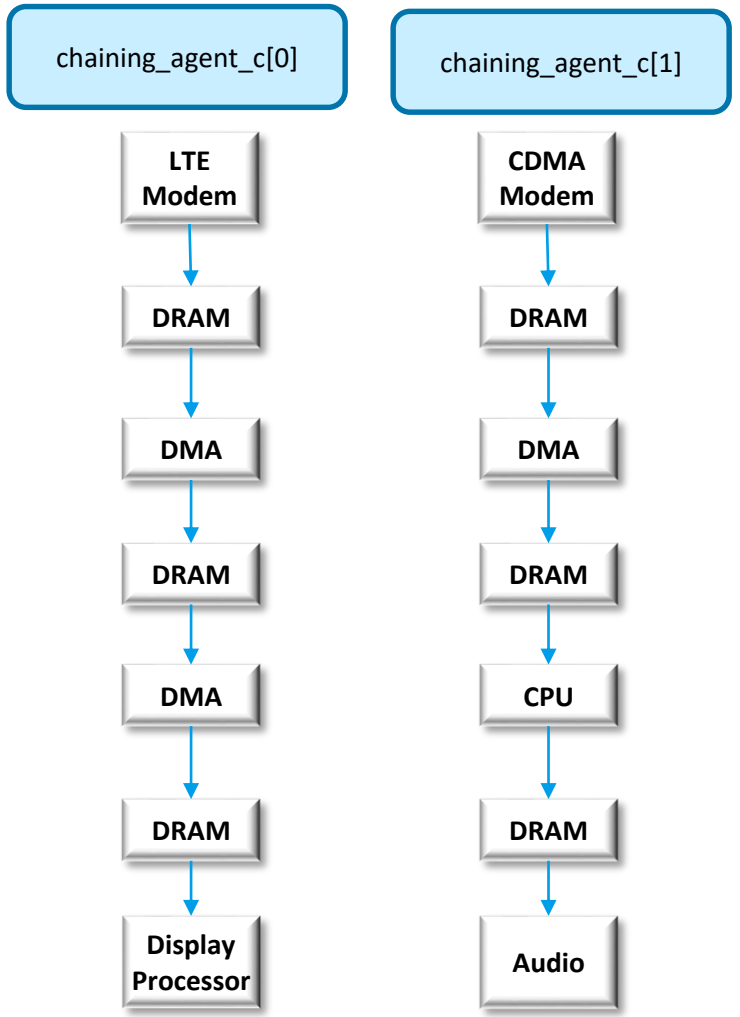
Sequential Chaining

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



Parallel Chaining

```
extend component pss_top {
  action entry {
    parallel {
      replicate (2) {
        do sequential_chaining_a;
      }
    }
  }
}
```



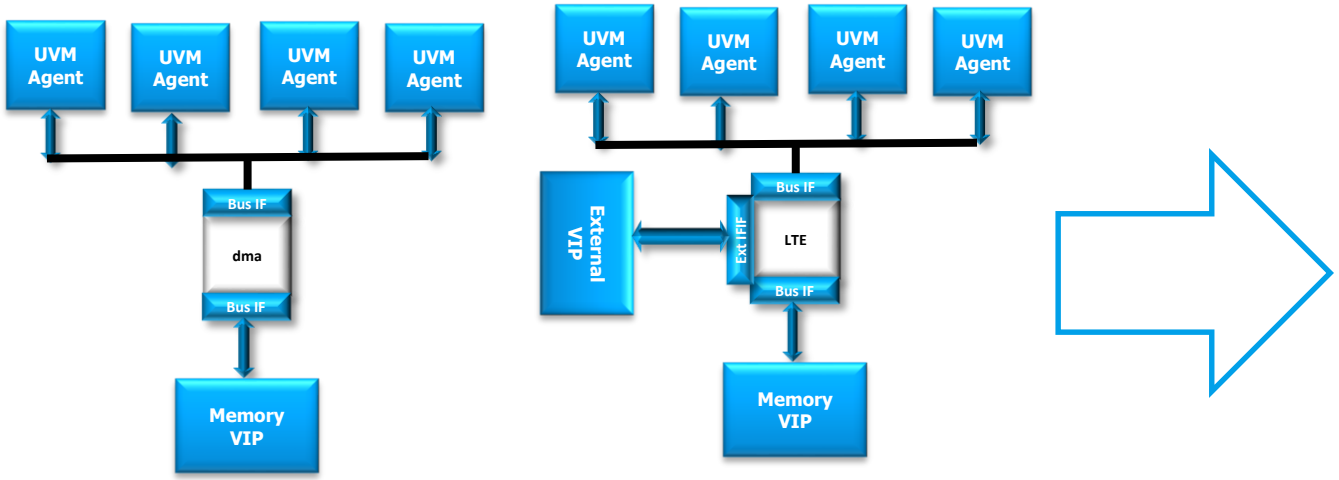
Chaining Combinations Coverage

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

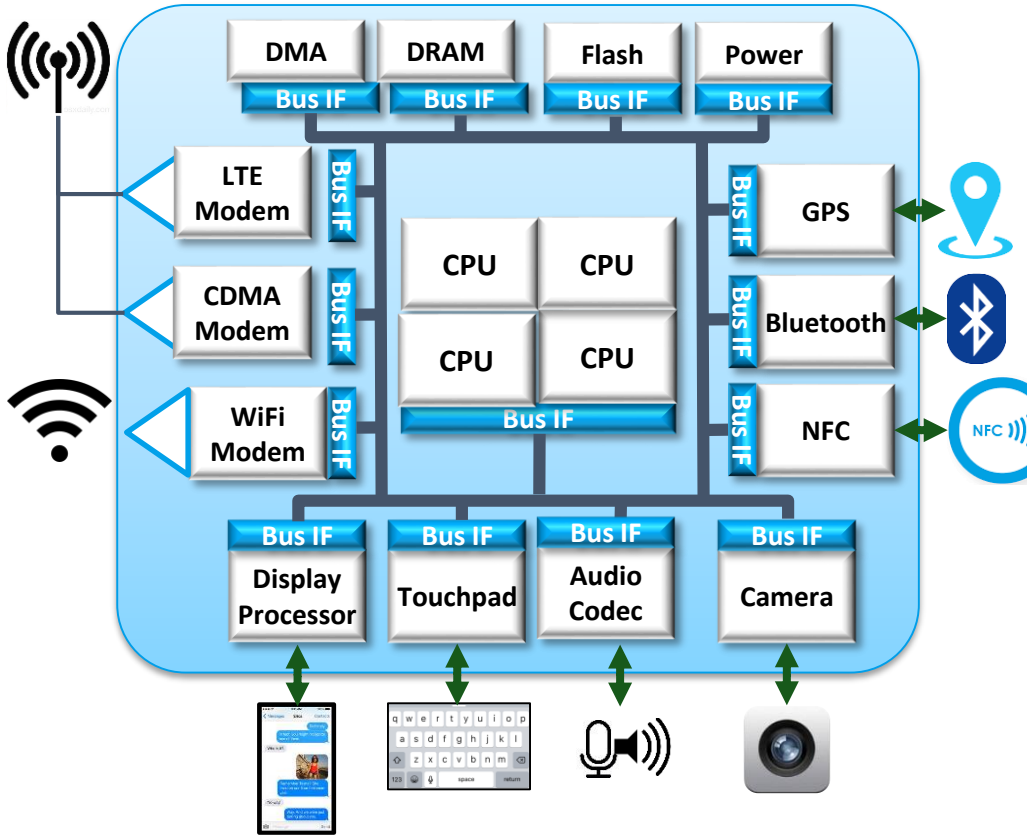
```
action sequential_chaining_a {
  rand source_e source_id;
  rand mem2mem_e mem2mem_id0, mem2mem_id1;
  rand sink_e sink_id;
  rand int size;
  activity {
    // Source
    select {
      [10] : do lte_source_a with {
               size == this.size;
               this.source_id == LTE_SOURCE;
             }
      [20] : do cdma_source_a with {
               size == this.size;
               this.source_id == CDMA_SOURCE;
             }
      [10] : do camera_source_awith {
               size == this.size;
               this.source_id == CAMERA_SOURCE;
             }
    }
    // Memory2Memory
    …
    // Sink
    …
  }
}
```

A Block-to-System Portability and Productivity

Verification productivity goal #3: Developing system level patterns to enable broad users to configure system level tests

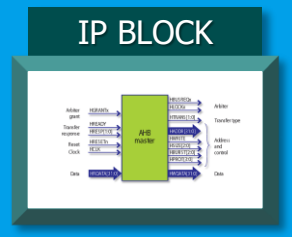
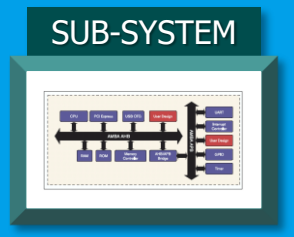
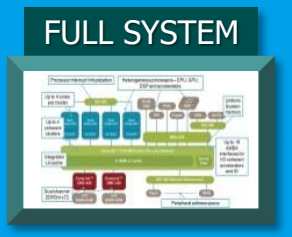


Block



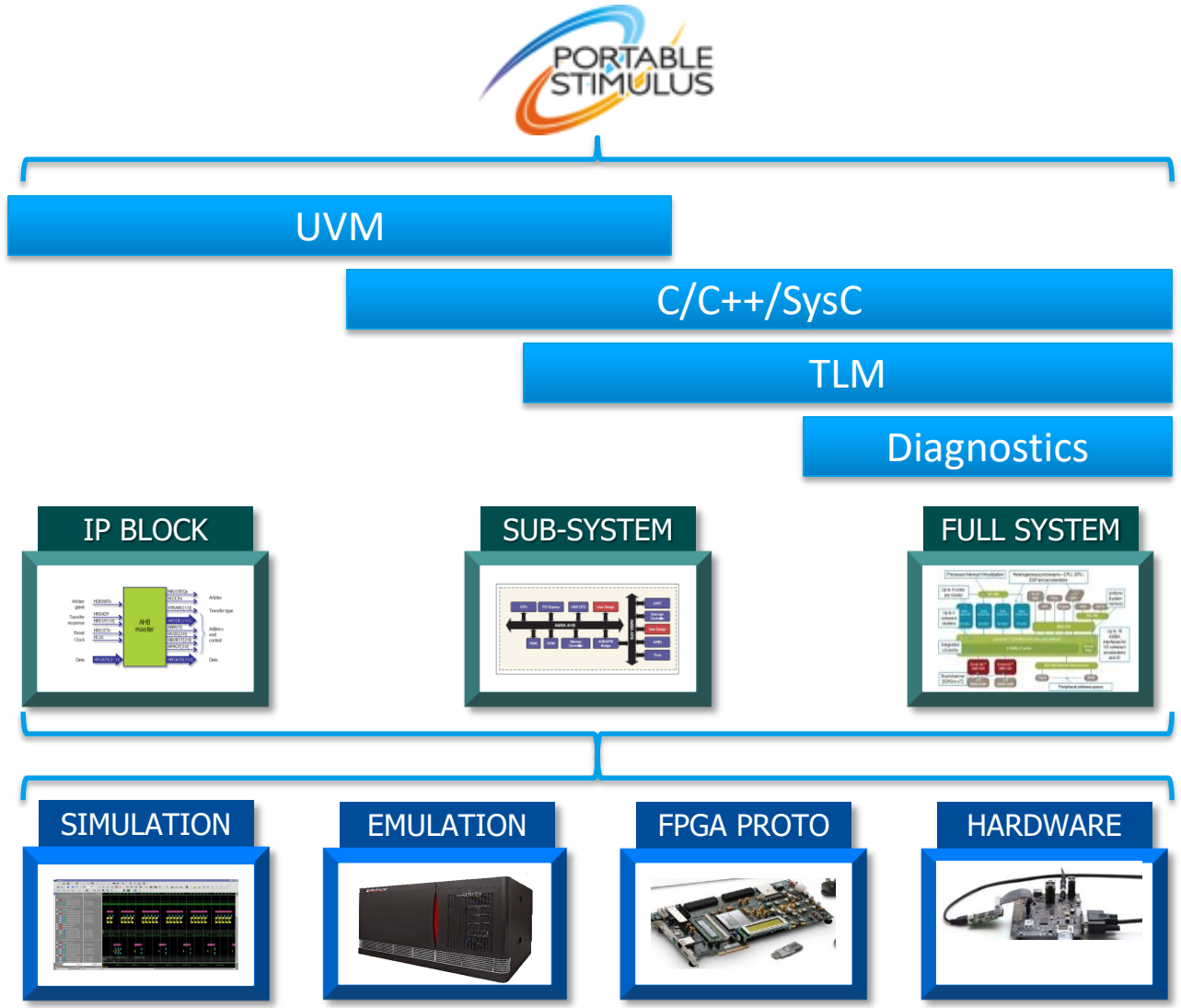
System

PSS Improves Individual Verification Phases

IP BLOCK	SUB-SYSTEM	FULL SYSTEM
		
Create block-level (UVM) tests & sequences based on scenario intent Easily compose complex, concurrent, high-coverage tests	Easily model system-level operations, reuse modular block level tests Drive hard-to-predict corner-case tests to flush out design operation	Generate software processor tests and IO transactions from common scenario model Flush out system functions with software-driven tests, while avoiding processor complexity

PSS Driving Methodology Portability

- Consistent specification of intent throughout process
- Methodology portability saves time & resource
- Shift-left enabler, parallel development of test content
- Test content reuse & scaling across verification program



2020 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Don't forget to join us for the lunch panel!

Q&A

THANK YOU!

accellera SYSTEMS INITIATIVE