Portable Test and Stimulus: The Next Level of Verification Productivity is Here

On behalf of the
Accellera Portable Stimulus Working Group

Tom Fitzpatrick, Mentor, a Siemens Business
Accellera PSWG Vice-Chair

Sharon Rosenberg, Cadence Design Systems

accellera SYSTEMS INITIATIVE

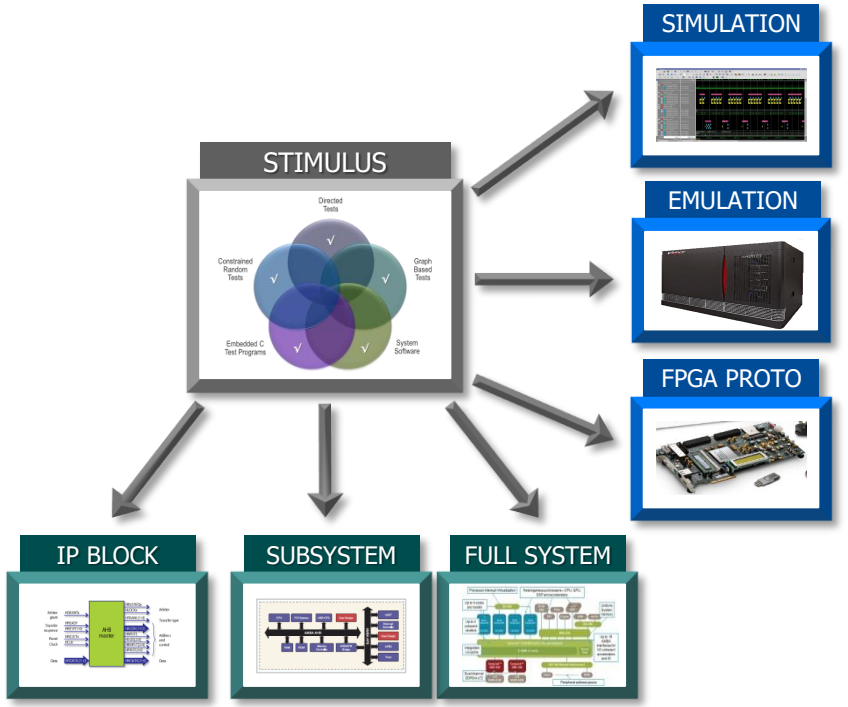
2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Agenda

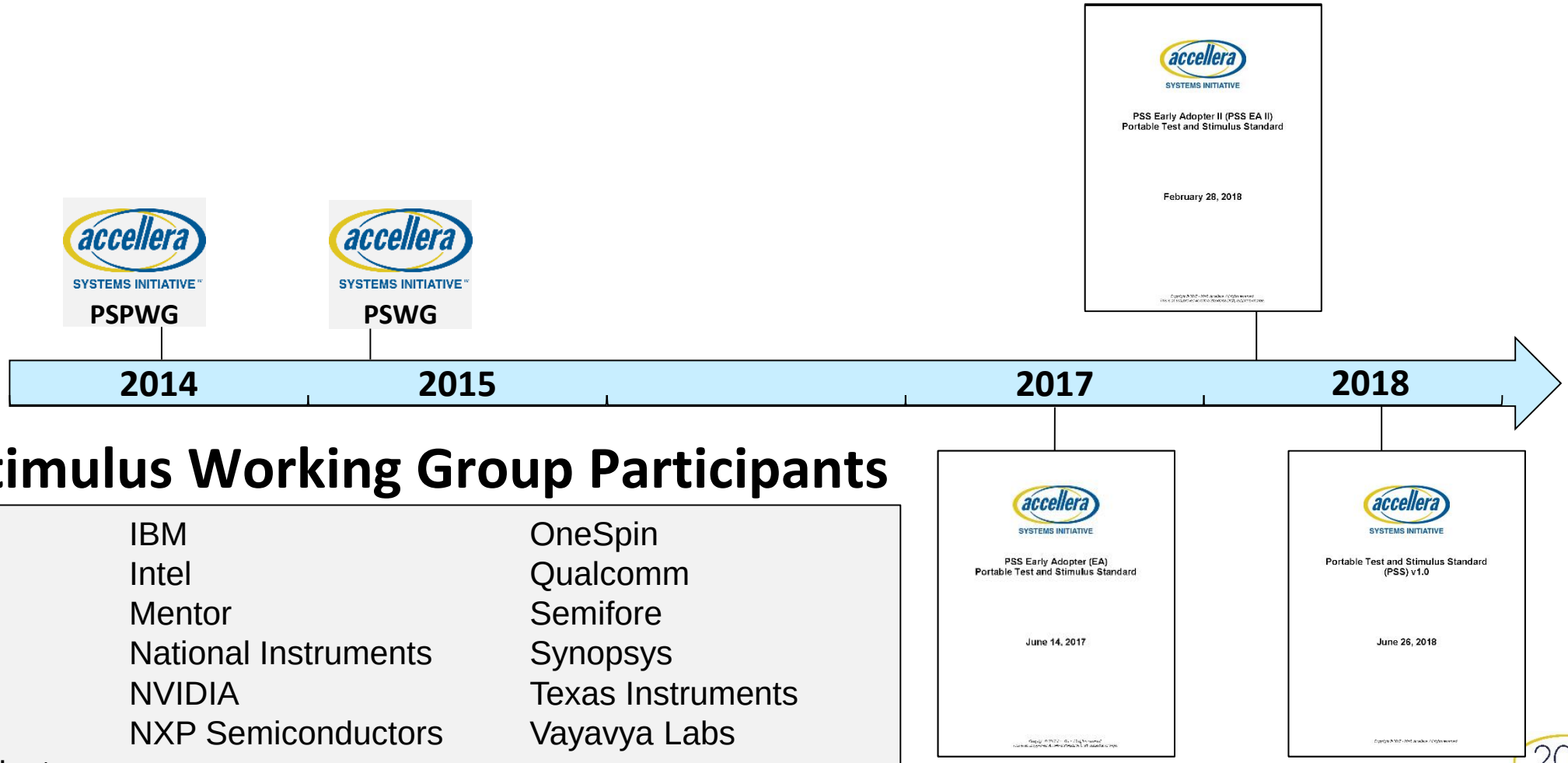
- Introduction: What and Why?
- “Hello World”: Language Concepts
- Block-to-System Example

It's an SOC World

- Design complexity continues to increase
  - Outstripping verification productivity
- System-level state space too big for effective UVM constrained-random
- Multiple verification platforms
- Need to reuse Test Intent
  - Higher abstraction
  - Block to system
  - Different design versions



The Portable Stimulus Journey

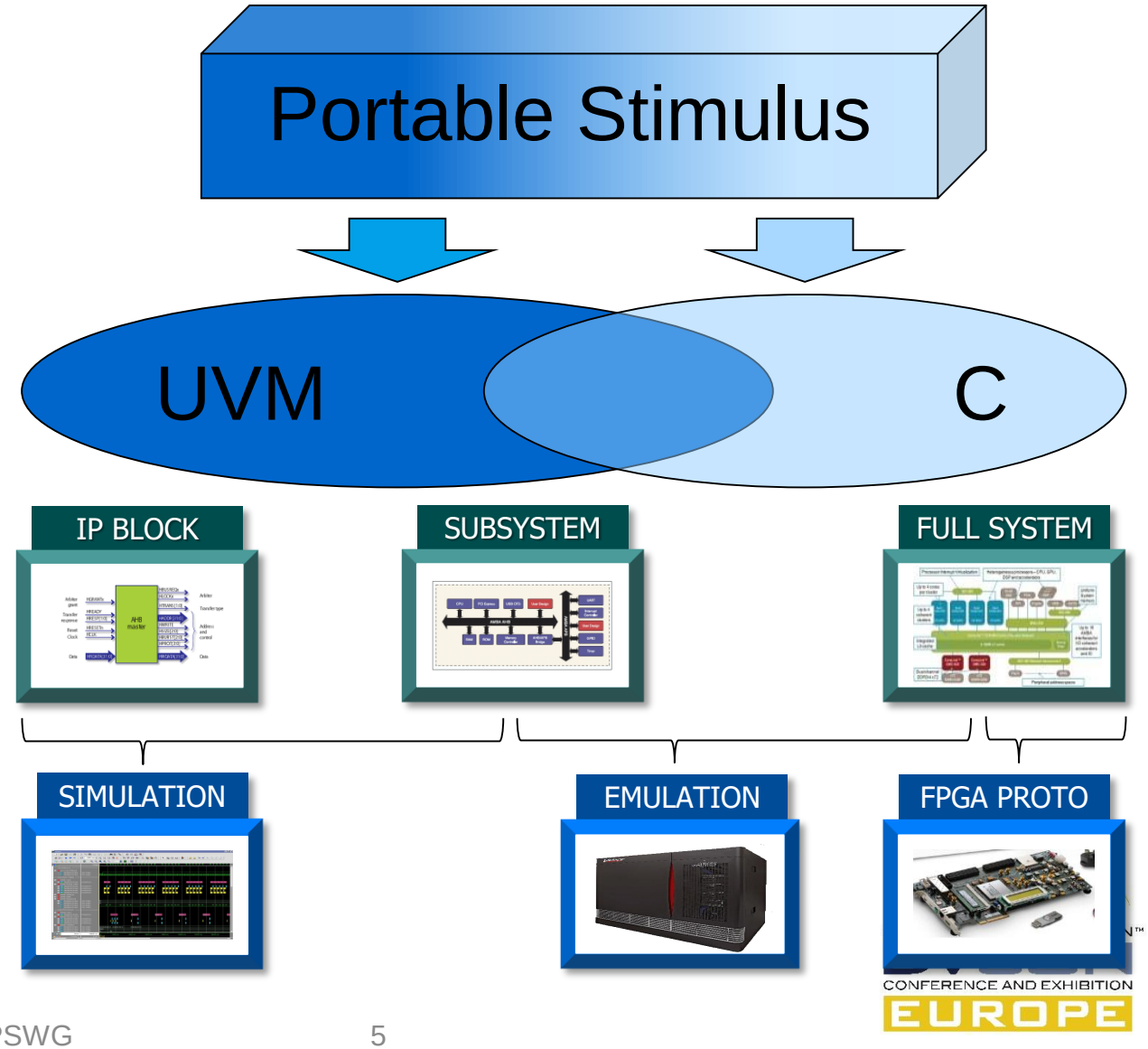


Portable Stimulus Working Group Participants

AMD	IBM	OneSpin
AMIQ EDA	Intel	Qualcomm
Analog Devices	Mentor	Semifore
Breker	National Instruments	Synopsys
Cadence	NVIDIA	Texas Instruments
Cisco	NXP Semiconductors	Vayavya Labs
Cypress Semiconductor		

Reuse of Test Intent Across Platforms/Users

- Single specification of test intent is critical
- **Constrain and randomize** at the **Scenario Level** by capturing:
  - interactions
  - dependencies
  - resource contention
- Abstraction lets tools automate test generation
  - Multiple targets
  - Target-specific customization

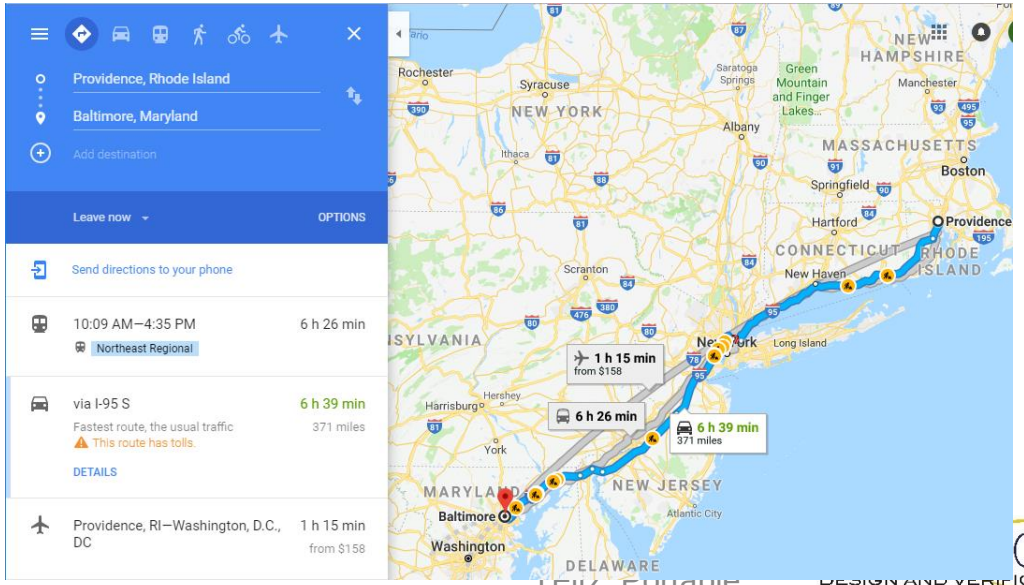


The Advantages of a Declarative Specification

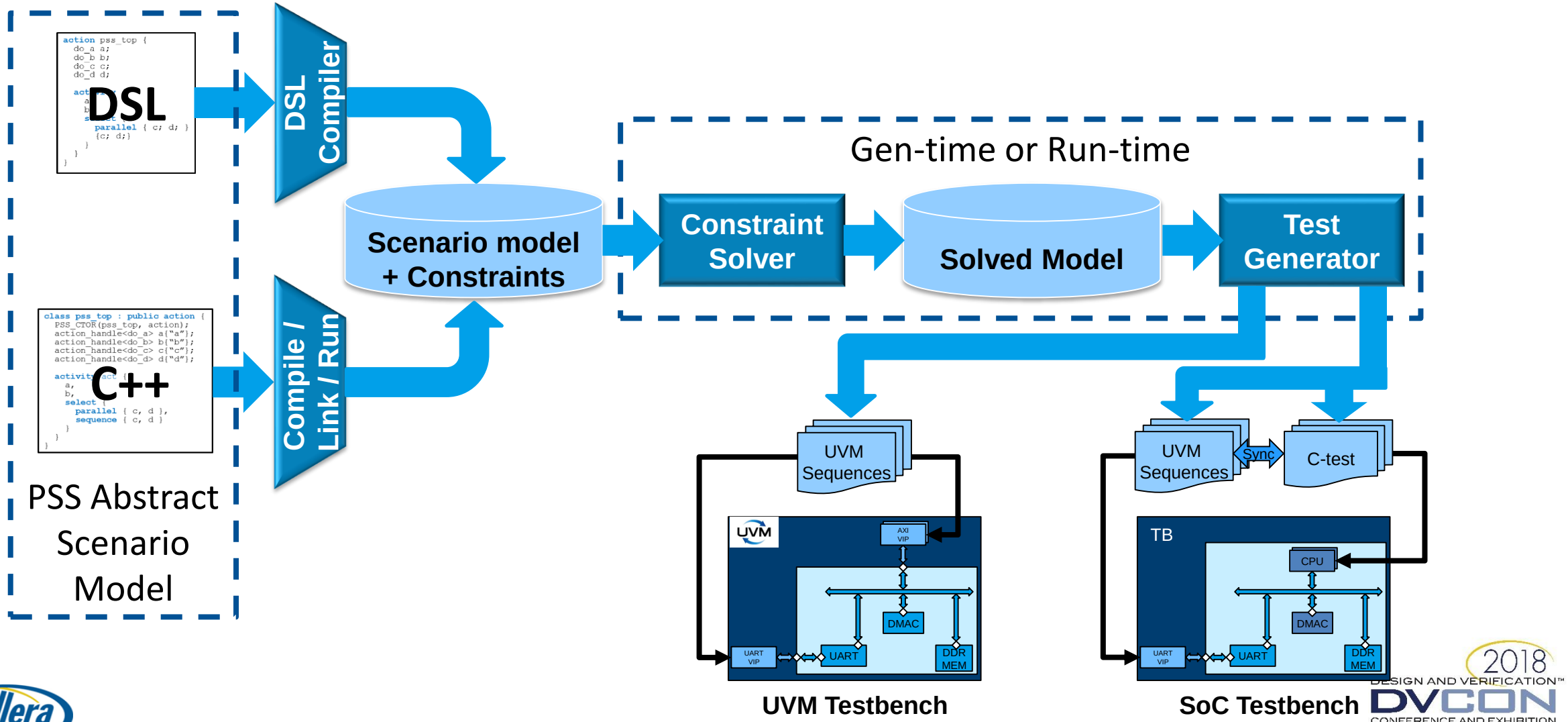
- Most SoC tests are directed
- Manually determining turn-by-turn directions
- Hard to account for new stops
- Route limited by driver's biases

- Declarative tests let the tool do the work
- Explore all possible options
- Easy to optimize
- Guided by preferences

- Declarative Portable Stimulus Description enables automation and analysis
- Automation makes test intent portable
- Enables retargeting to different environments
- Automation makes test techniques portable
- Bring automated constraint-driven tests to SoC



Projected Tool Flow



What Portable Stimulus Is NOT

- **NOT** a UVM replacement
- **NOT** a reference implementation
- **NOT** one forced level of abstraction
  - Expressing intent from different perspectives is a primary goal
- **NOT** Monolithic
  - Representations would typically be composed of portable parts
- **NOT** Two standards
  - PSS/DSL and PSS/C++ input formats describe 1:1 semantics
  - Tools shall consume both formats
- **NOT** Just stimulus
  - Models Verification Intent
  - Stimulus, checks, coverage, scenario-level constraints
  - Portable test realization

"HELLO WORLD": LANGUAGE CONCEPTS

Hello World: Atomic Actions

hello world

component groups elements for *reuse and composition*

action defines *behavior*

exec defines *implementation*

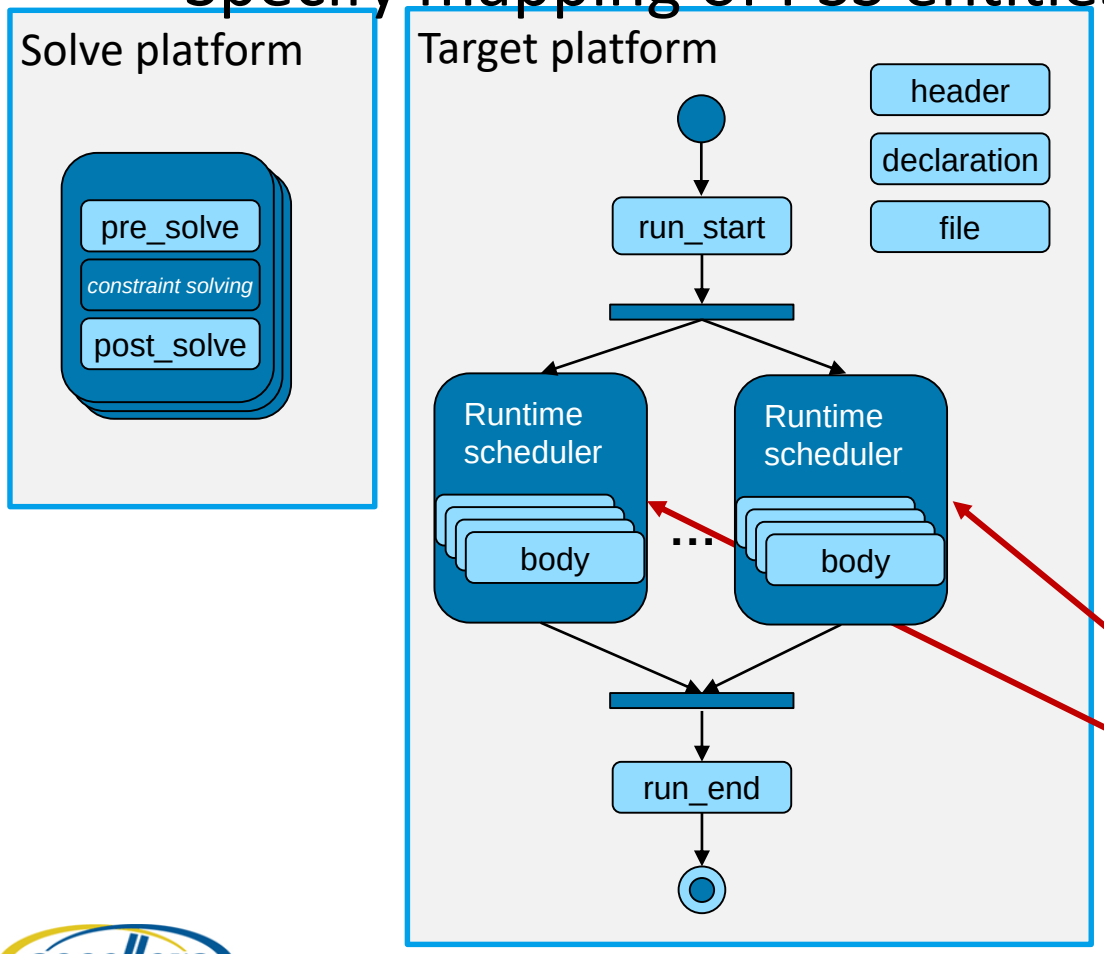
```
component pss_top {
  action hello_world_a{
    exec body SV = """
      $display("Hello World");
    """;
  }
}
```

```
class hello_world_a_seq_1 extends uvm_sequence;
      `uvm_object_utils(hello_world_a_seq_1)

      virtual task body();
            $display("Hello World");
      endtask
endclass
```

- ✓ Reuse
- ✓ Composition
- ✓ Abstract behaviors
- ✓ Retargetable Implementations

Exec Block Types

- Specify mapping of PSS entities to their implementation



test.c

```
#include <stdint.h>

void declared_func() {
…
}

void test_main() {
  do_run_start();
  fork_threads();
  do_run_end();
}

void thread0() {
…
// step N
  do_body();
…
};

void thread1() {
…
}
```

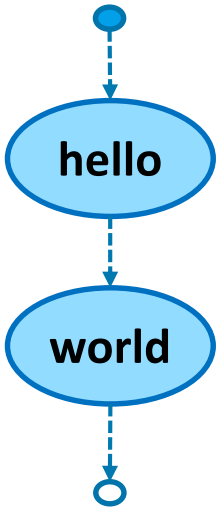
test.sh

```
gcc –c test.c –DBARE_METAL
```

Could be SV or other language

Could be multiple threads on one core, or threads running on different cores

Hello World: Compound Actions



compound action
traverses *other actions*

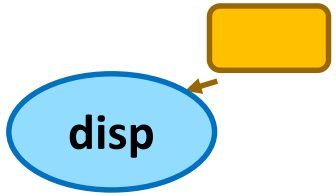
activity
defines *scheduling*

```
action hello_world_a {
  hello_a h;
  world_a w;
  activity {
    h;
    w;
  }
}
```

```
void hello_world_a_test_1() {
  printf ("Hello\n");
  printf ("World\n");
}
```

- ✓ Behavior encapsulation
- ✓ Behavior scheduling

Hello World: Data Flow Objects




```
component pss_top {
  buffer msg_buf {
    rand string s;
  }


  action display_a {
    input msg_buf msg;
    exec body SV = """
      $display("{{msg.s}}");
    """;
  }
```

buffer defines *data flow*

stream and **state** also defined

data may be *randomized*

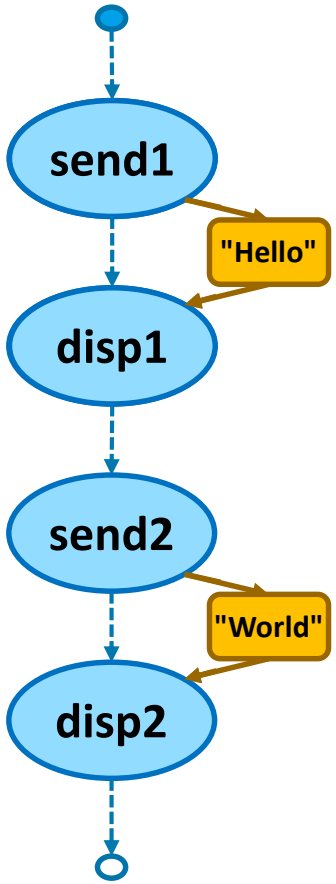
input defines *flow requirement*

output too

"moustache" passes model elements to templates

- ✓ **Complex data structures**
- ✓ **Data flow modeling**
- ✓ **Constrained random data**
- ✓ **Reactivity**

Hello World: Data Flow Objects

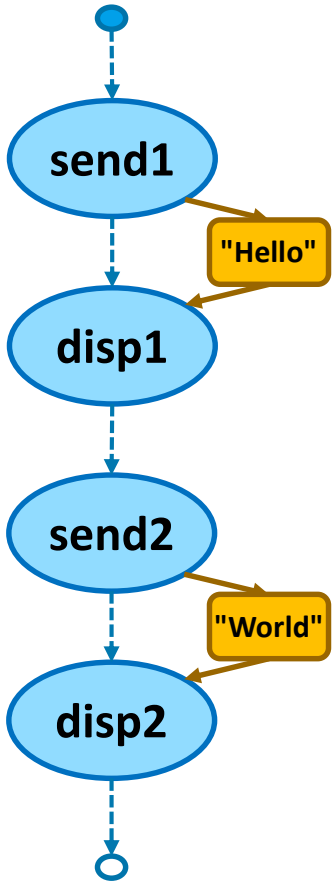



```
component pss_top {
  buffer msg_buf {
    rand string s;
  }

  action display_a {
    input msg_buf msg;
    exec body SV = """
      $display("{{msg.s
    """;
  }
```

```
  action send_a {
    output msg_buf msg;
  }
  action hello_world_a {
    send_a send1, send2;
    display_a disp1, disp2;
    activity {
      send1;
      disp1 with {msg.s == "Hello ";};
      send2;
      disp2 with {msg.s == "World";};
      bind send1.msg disp1.msg;
      bind send2.msg disp2.msg;
    }
  }
}
```

- ✓ Directed testing when desired
- ✓ In-line constraints

Hello World: Packages



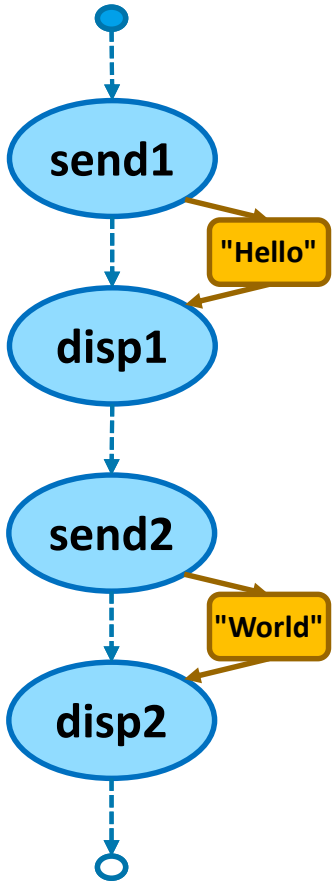

```
package hw_pkg;
  buffer msg_buf {
    rand string s;
  }
}
component pss_top {
  import hw_pkg::*;
  action display_a {
    input msg_buf msg;
    exec body SV = """
      $display("{{msg.s}}");
    """;
  }
```

```
on send_a {
utput msg_buf msg;

ion hello_world_a {
end_a send1, send2;
isplay_a disp1, disp2;
ctivity {
   send1;
   disp1 with {msg.s == "Hello ";};
   send2;
   disp2 with {msg.s == "World";};
   bind send1.msg disp1.msg;
   bind send2.msg disp2.msg;
```

```
    }
  }
}
```

✓ Additional reuse and encapsulation

Hello World: Inferred Actions

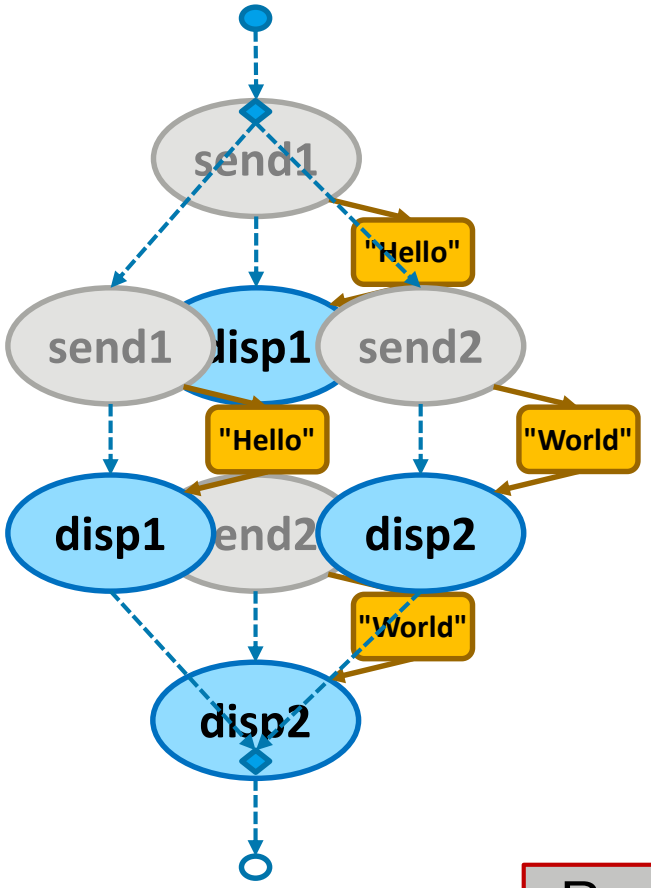



```
package hw_pkg {
  buffer msg_buf {
    rand string s;
  }
}
component pss_top {
  import hw_pkg::*;
  action display_a {
    input msg_buf msg;
    exec body SV = """
      $display("{{msg.s
    """;
  }
```

```
  action send_a {
    output msg_buf msg;
  }
  action hello_world_a {
    send_a send1, send2;
    display_a disp1, disp2;
    activity {
        send1;
        disp1 with {msg.s == "Hello ";};
        send2;
        disp2 with {msg.s == "World";};
    }
  }
}
```

✓ Abstract partial specifications

Hello World: Activity Statements




```
package hw_pkg {
  buffer msg_buf {
    rand string s;
  }
}
component pss_top {
  import hw_pkg::*;
  action display_a {
    input msg_buf msg;
    exec body SV = """
      $display("{{msg.s
    """;
  }
```

```
action send_a {
    output msg_buf msg;
  }
  action hello_world_a {

    display_a disp1, disp2;
    activity {
      select {
        disp1 with {msg.s == "Hello ";};

        disp2 with {msg.s == "World";};


      }
    }
}
```

Randomly choose a branch

✓ Scenario-level randomization

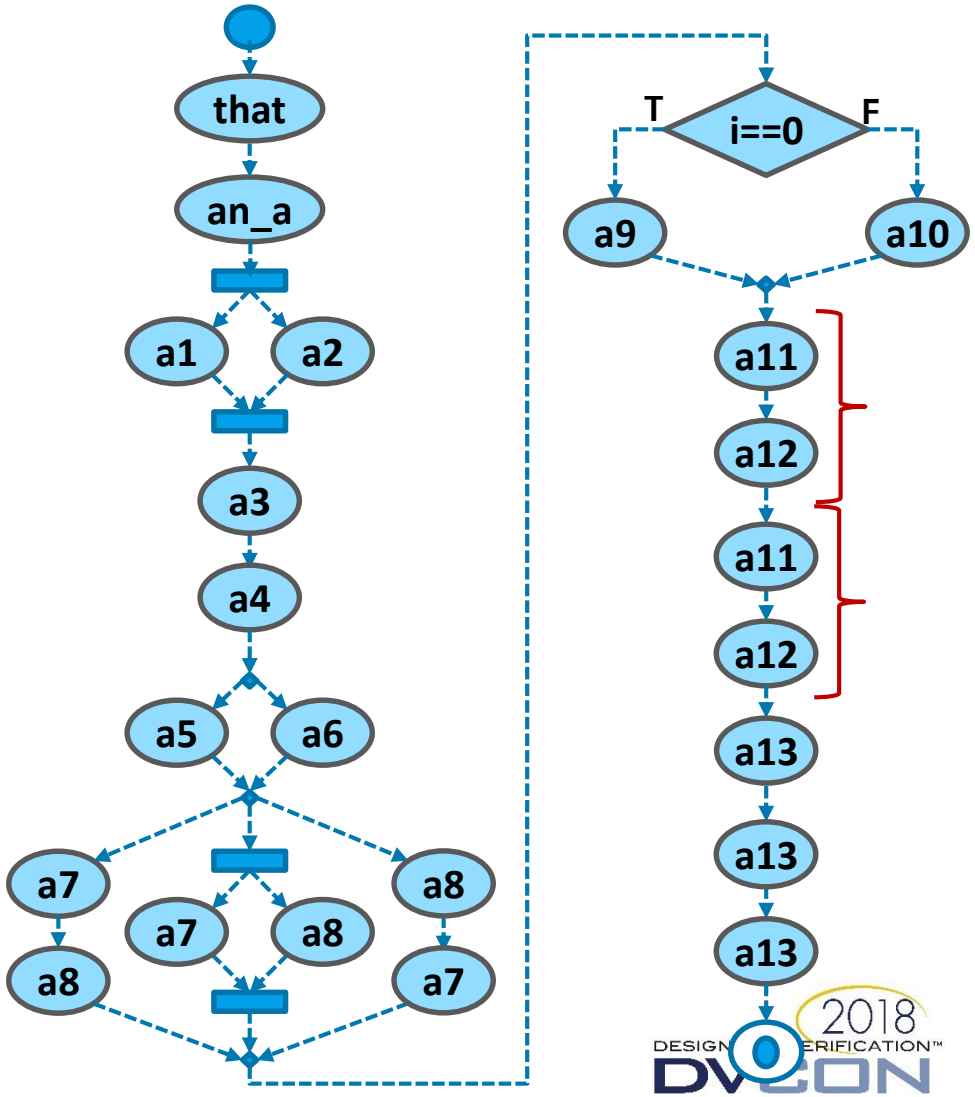
Activity: Robust Expression of Critical Intent

```
activity {
  that;
  do an_a;
  parallel {a1, a2};
  sequence {a3, a4};
  select {a5, a6};
  schedule {a7, a8};
  if (i == 0) {a9;}
  else {a10;}
  repeat (2) {a11, a12};
  foreach (arr[j]) {
    a13 with {a13.val == arr[j];};
}
```

Action instance *traversal*

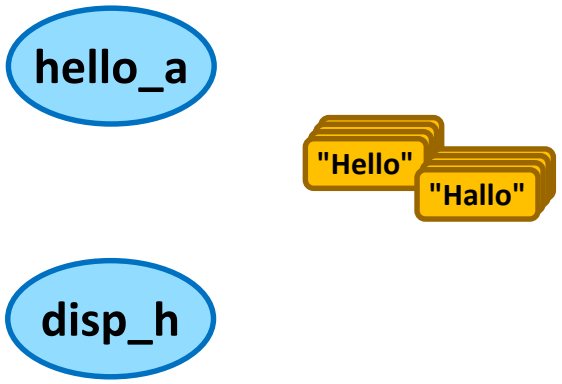
Anonymous action *traversal*

Subject to flow/resource constraints



✓ **Robust scheduling support**

Hello World: Extension & Inheritance



Type extension

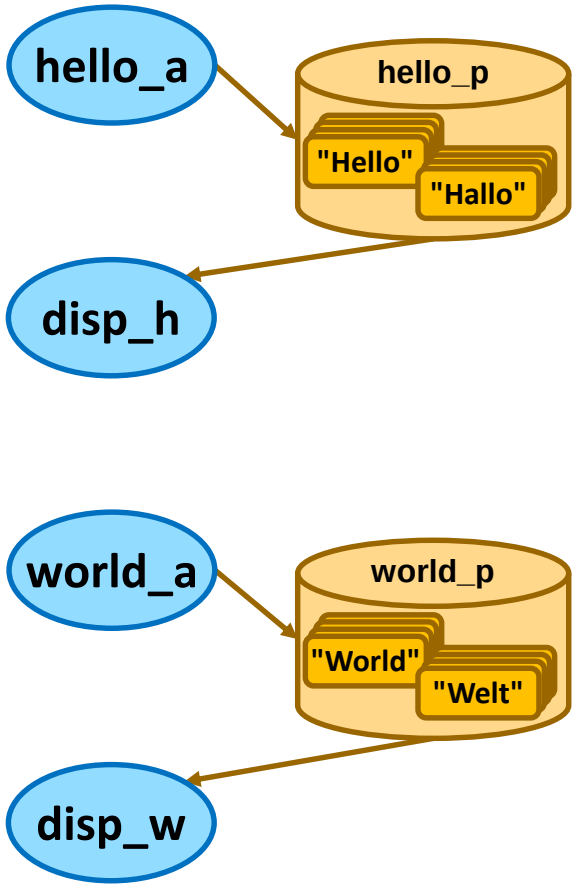
Inheritance

```
extend component pss_top {
  buffer hello_buf : msg_buf {
    constraint {msg.s in["Hello", "Hallo"];}
  }
  action disp_h : display_a {
    override {type msg_buf with hello_buf};
  }
  action hello_a {
    output hello_buf msg;
  }
}
```

Override

- ✓ Type extension (aspect-oriented programming)
- ✓ Object-oriented inheritance
- ✓ Type (& instance) override

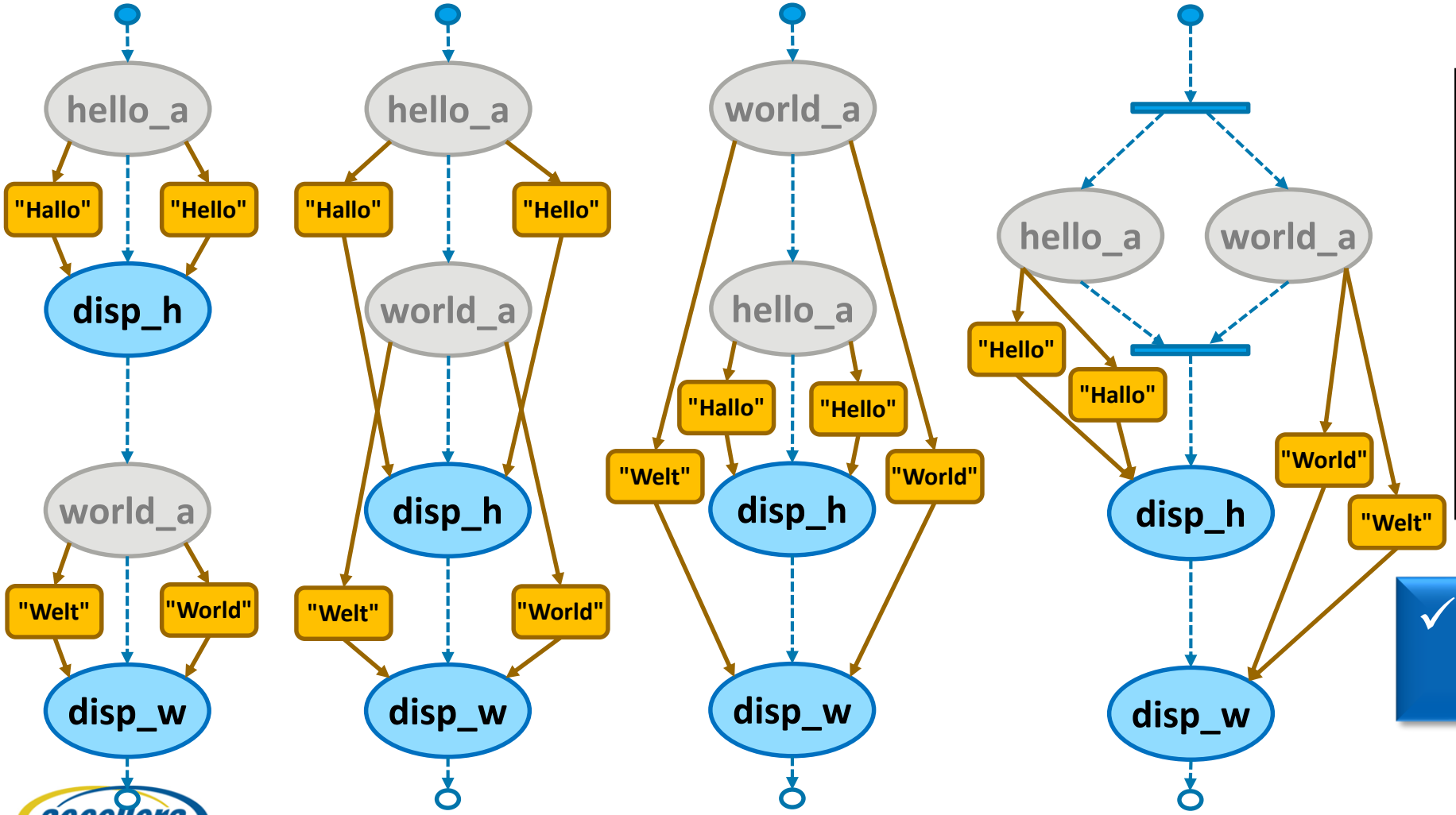
Hello World: Object Pools & Binding




```
extend component pss_top {
  buffer hello_buf : msg_buf {
    constraint {msg.s in["Hello", "Hallo"];}
  }
  action disp_h : display_a {
     override {type msg_buf with hello_buf};
  }
  action hello_a {
    output hello_buf msg;
  }
  pool hello_buf hello_p;
  bind hello_p *;
}
```

- ✓ Constrain data paths
- ✓ Preserve intent

Hello World: Scenarios




```
action hello_world_a {
  activity {
    sequence {
      do disp_h;
      do disp_w;
    }
  }
}
```

anonymous action traversal

✓ Multiple scenarios from simple specification

Hello World: C++

```
package hw_pkg {


    buffer msg_buf {

        rand string s;
    }
}
```

```
class hw_pkg : public package {
  PSS_CTOR(hw_pkg, package);

  struct msg_buf : public buffer {
    PSS_CTOR(msg_buf, buffer);
    rand_attr<std::string> s {"s"};
  };
};
type_decl<hw_pkg> hw_pkg_decl;
```

Hello World: C++

```
component pss_top {
  import hw_pkg::*;

  action display_a {

    input msg_buf msg;
    exec body SV = """
      $display("{{msg.s}}");
    """;
  }

  action send_a {

    output msg_buf msg;
  }
```

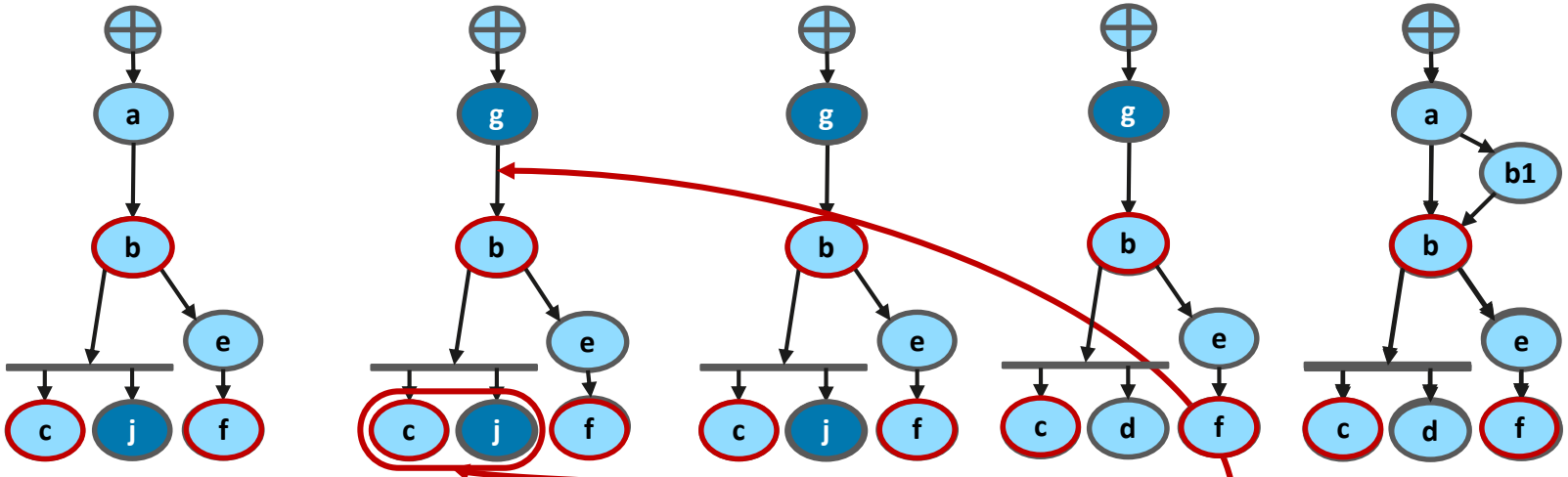
```
class pss_top : public component {
  PSS_CTOR(pss_top, component);

  class display_a : public action {
    PSS_CTOR(display_a, action);
    input <hw_pkg::msg_buf> msg {"msg"};
    exec e {exec::body, "SV",
            "$display(\"{{msg.s}}\");"};
  };
  type_decl<display_a> display_a_decl;

  class send_a : public action {
    PSS_CTOR(send_a, action);
    output <hw_pkg::msg_buf> msg {"msg"};
  };
  type_decl<send_a> send_a_decl;
```

Hello World: C++

```
  pool msg_buf msg_p;
  bind msg_p *;
  action hello_world_a {

    display_a disp1, disp2;

    activity {
      select {
        disp1 with {msg.s == "Hello ";
        disp2 with {msg.s == "World";}
    }
  }
}
```

```
  pool <hw_pkg::msg_buf> msg_p {"msg_p"};
  bind b {msg_p};
  class hello_world_a : public action {
    PSS_CTOR(hello_world_a, action);
    action_handle<display_a> disp1 {"disp1"},
                             disp2 {"disp12"};
    activity a {
      select {
        disp1.with (disp1->msg->s == "Hello"),
        disp2.with (disp2->msg->s == "World")
      }
    };
  };
  type_decl<hello_world_a> hello_world_a_decl;
};
type_decl<pss_top> pss_top_decl;
```

Solution Space Mapping



Partial Specifications are ***Flexible***

```
action test_top {
  do_a a; do_b b;
  do_c c; do_d d;
  do_e e; do_f f;

  activity {
    a;
    b;
    select {
      parallel { c; d; }
      {e; f;}
    }
  }
}
```

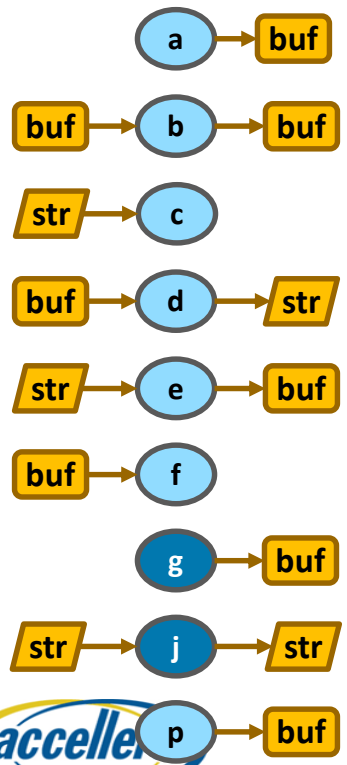
```
action test_top {
          do_b b;
  do_c c;
          do_f f;

  activity {

    b;
    select {
                  c;
       f;
    }
  }
}
```

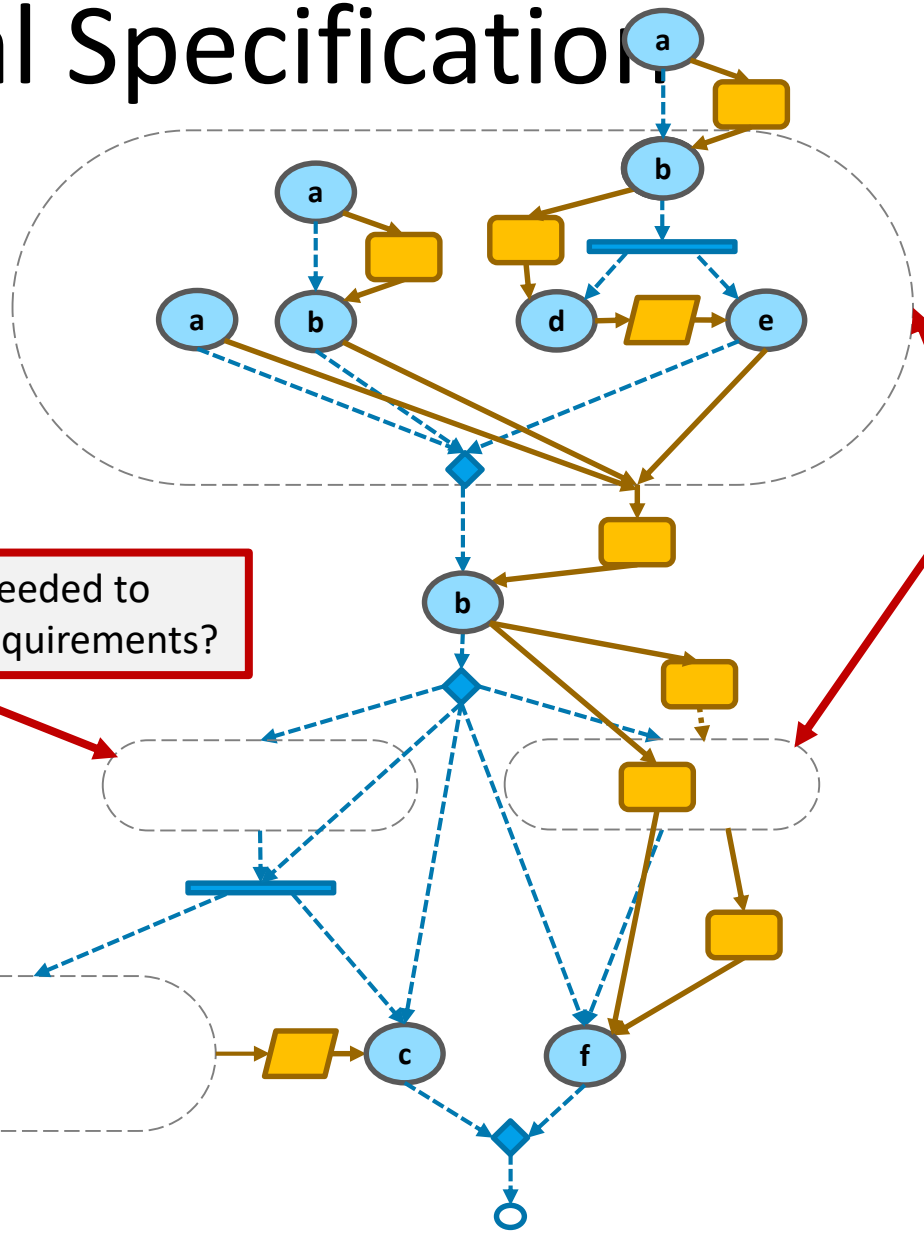
```
buffer mbuf {…};

action do_a {
  output mbuf m;
  …};

action do_b {
  input mbuf m;
  output mbuf o;
  …};

action do_g {
  output mbuf m;
  …};
```

```
stream mstr {…};

action do_c {
  input mstr s;
  …};

action do_d {
  output mstr s;
  …};

action do_j {
  output mstr m;
  …};
```

Resolving a Partial Specification

```
action test_top {

  activity {
    b;
    select {
      c;
      f;
    }
  }
}
```





What set of actions is needed to support downstream requirements?

What set of actions will produce a **stream** of the correct type?

What combination of known actions will produce a **buf** of the correct type?

Are there any **resource** conflicts that constrain the possible scheduling?

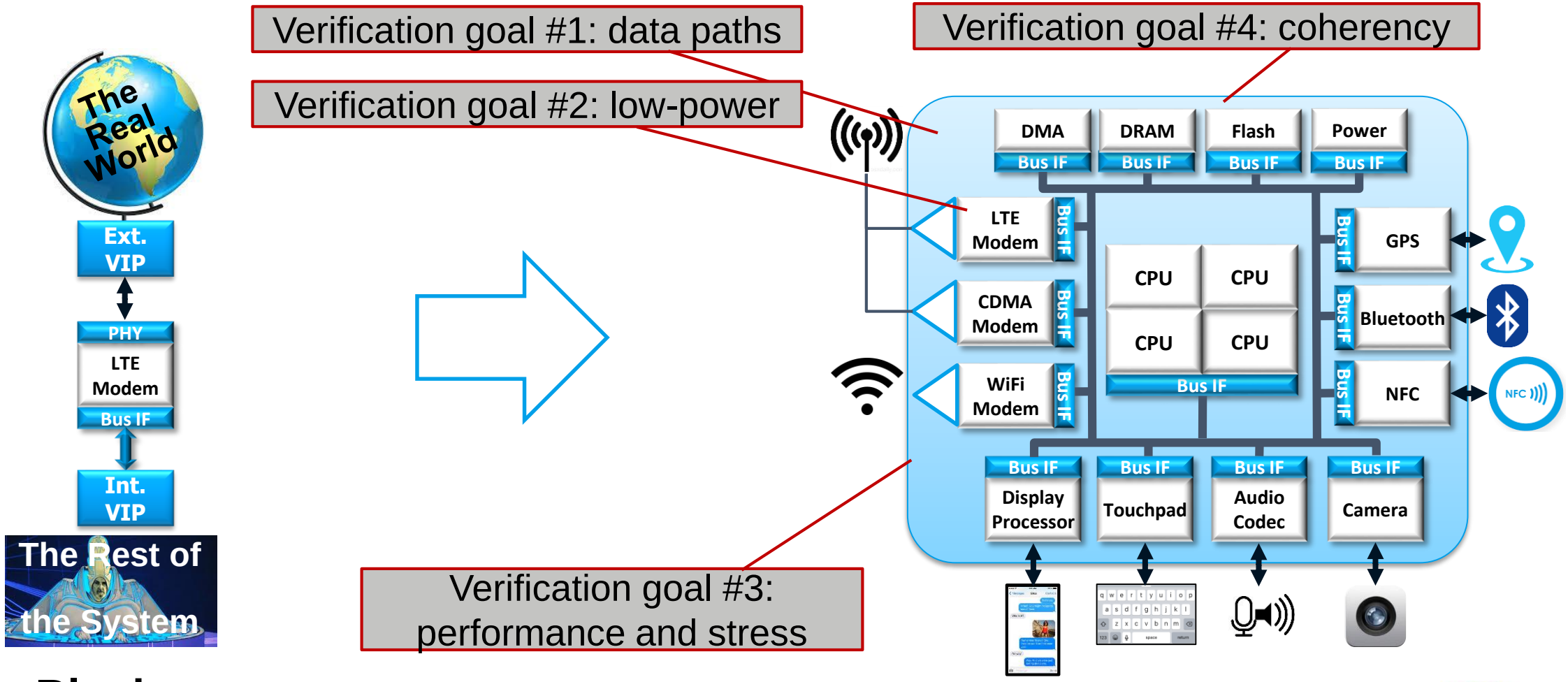
A Quick Recap: PSS Gives You…

- ✓ Reuse
- ✓ Composition
- ✓ Abstract behaviors
- ✓ Retargetable Implementations
- ✓ Behavior encapsulation
- ✓ Behavior scheduling
- ✓ Complex data structures
- ✓ Data flow modeling
- ✓ Constrained random data
- ✓ Reactivity
- ✓ Directed testing when desired
- ✓ In-line constraints
- ✓ Additional reuse and encapsulation
- ✓ Abstract partial specifications
- ✓ Scenario-level randomization
- ✓ Robust scheduling support
- ✓ Type extension
- ✓ Object-oriented inheritance
- ✓ Type (& instance) override
- ✓ Constrain data paths
- ✓ Preserve intent
- ✓ Multiple scenarios from simple specification

But wait! There's more!

BLOCK-TO-SYSTEM EXAMPLE

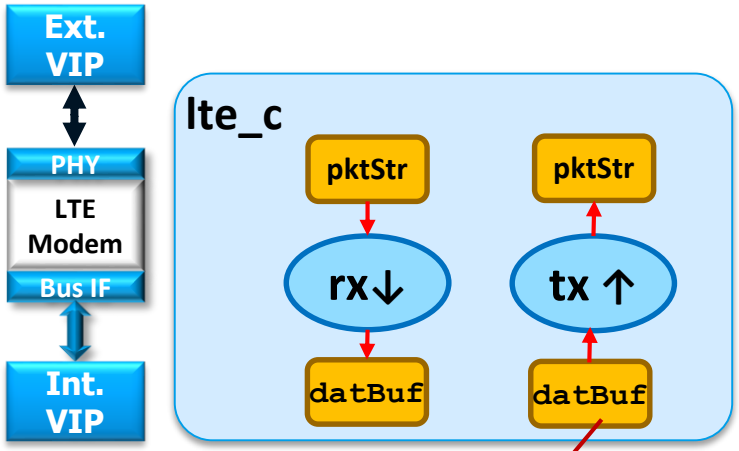
A Block-to-System Example



Block **System**

Decomposing the scenarios into actions allows reusable library of building blocks

The LTE Modem Component



For modeling actions read the spec and identify key behaviors

```
component lte_c {
  action rx {
    input pktStr pStr;
    output datBuf dBuf;
    constraint {pStr.size < 200;}
  }
  action tx {
    input datBuf dBuf;
    output pktStr pStr;
    constraint {pStr.size < 200;}
  }
}
```

Inputs and output should not assume the provider of the information

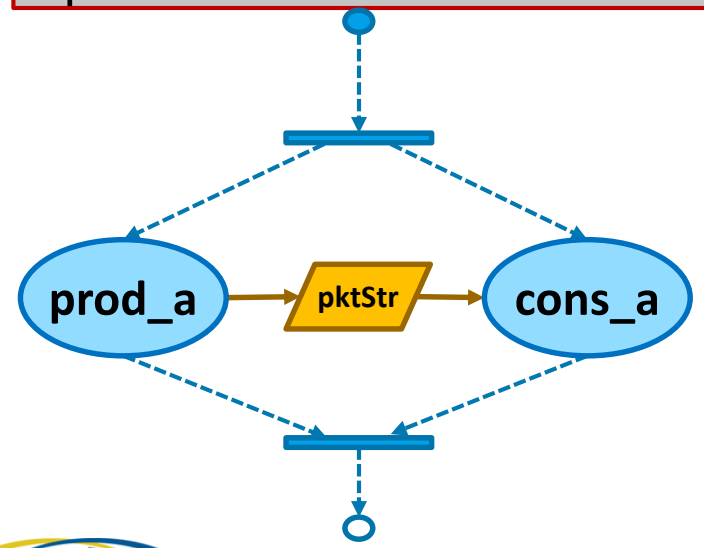
Rules of correct inputs and outputs should be captured in constraints

Define Data Flow Objects

buffer requires *sequential* producer-consumer execution

rand fields are randomized

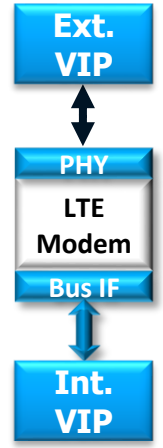
stream requires *parallel* producer-consumer execution



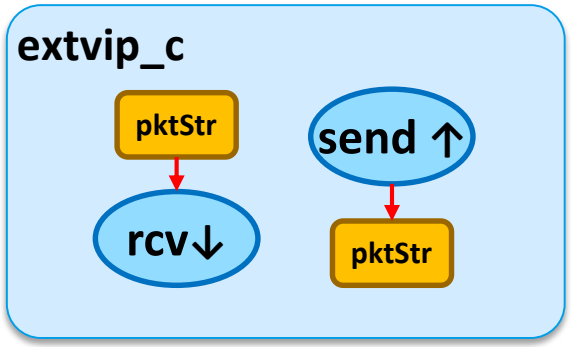

```
package data_flow_pkg{
  enum dir_e {inb=0, outb};
  enum kind_e {video, text, msg};
  buffer datBuf {
    rand dir_e dir;
    rand bit [7:0] length;
    rand bit [31:0] addr;
    rand kind_e kind;
  }
  stream pktStr {
    rand dir_e dir;
    rand bit [15:0] size;
    bit [47:0] MAC_src;
    bit [47:0] MAC_dst;
    rand kind
  }
}
```

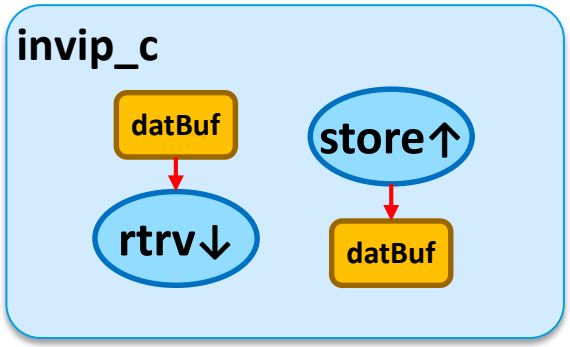
PSS provides packages for code reuse

```
component lte_c {
  import data_flow_pkg::datBuf;
  …
}
```



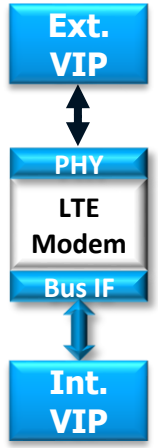
The VIP Components



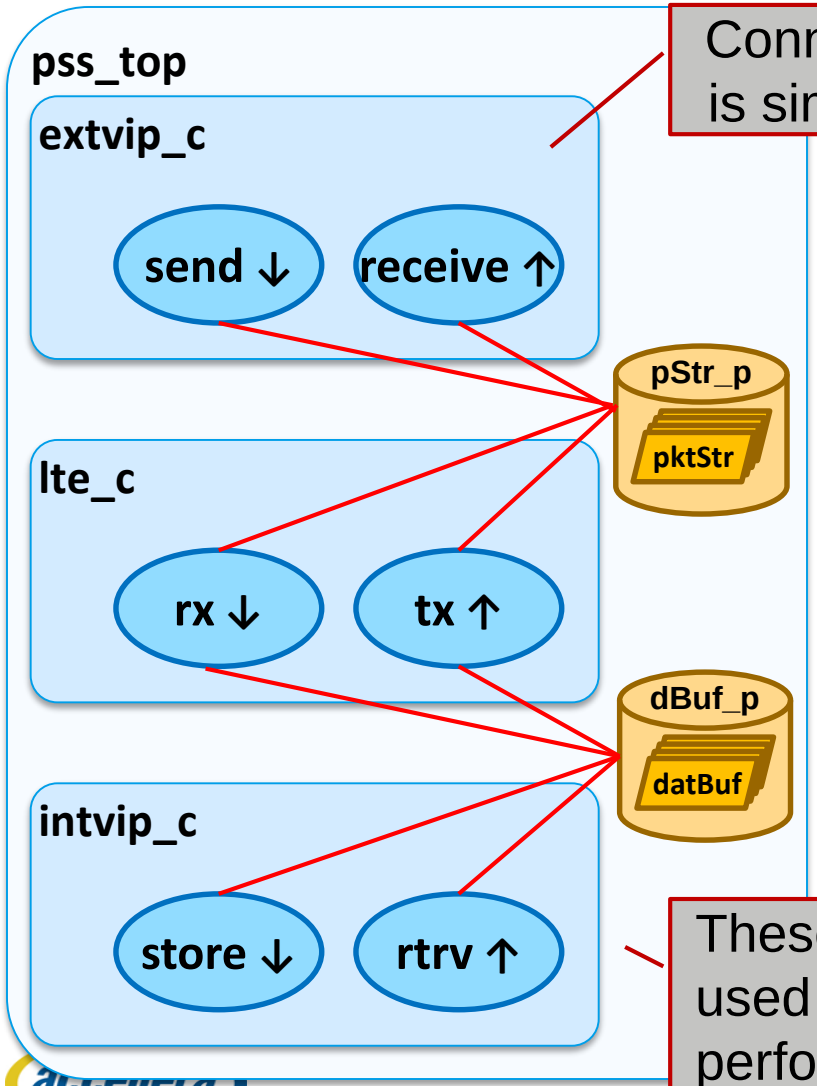



```
component extvip_c {
  action send {
    output pktStr pStr;
    constraint {pStr.size > 100;}
  }
  action receive {
    input pktStr pStr;
    constraint {pStr.size > 100;}
  }
}
```

```
component intvip_c {
  action store {
    output datBuf dBuf;
  }
  action rtrv {
    input datBuf dBuf;
  }
}
```



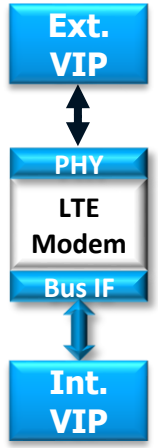
PSS Model Instantiation



Connection of new components is simple – just instantiate them

predefined component pss_top

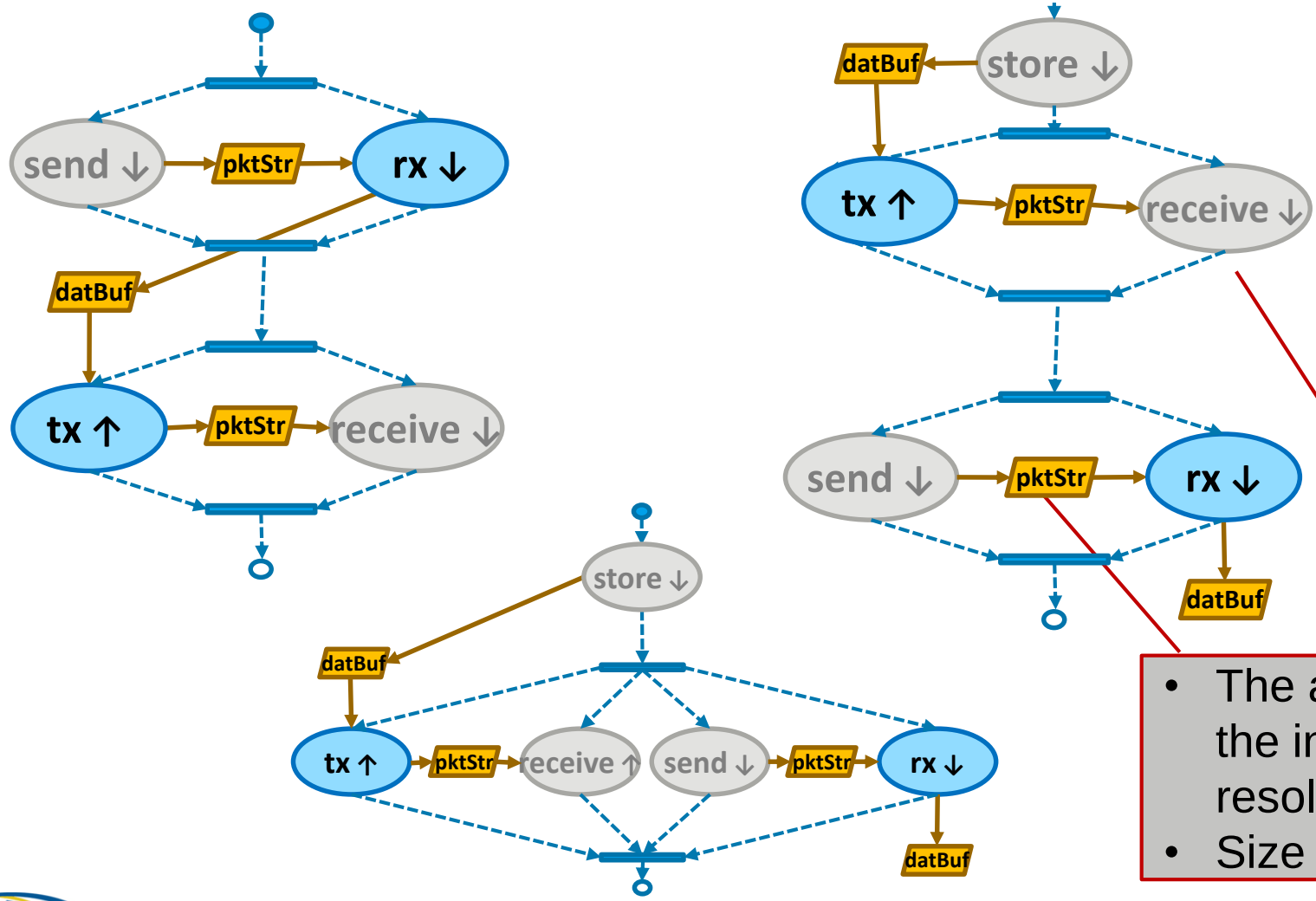
```
component pss_top {
  import data_flow_pkg::*;

  pool pktStr pStr_p;
  bind pStr_p {xvip.*, lte.*};
  pool datBuf dBuf_p;
  bind dBuf_p {ivip.*, lte.*};

  extvip_c xvip;
  lte_c lte;
  intvip_c ivip;

}
```



These two actions can be used for coherency, stress, performance and more

- Now we are ready to create portable scenarios!
- Endless number of scenarios can be randomized on this model

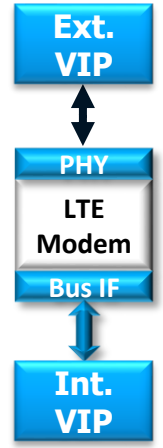
Portable Scenario Example




```
action my_scenario {
  activity {
    schedule {
      do lte_c::rx;
      do lte_c::tx;
    }
  }
}
```

- The scenarios are focus on the intent

- The algebraic constraints of the input and the output are resolved
- Size < 200 && size > 100



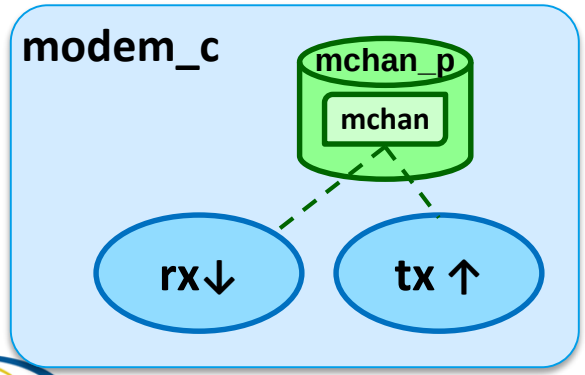
The Modem Component + Resources

What if the Modem is half-duplex?
i.e it is illegal to run rx & tx actions at the same time

`resource` defines a *resource object*

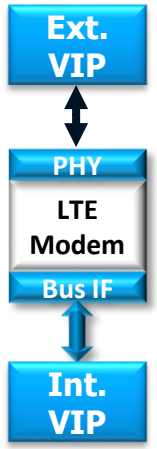
`pool` defaults to *size == 1*

`lock` declares *exclusive access*



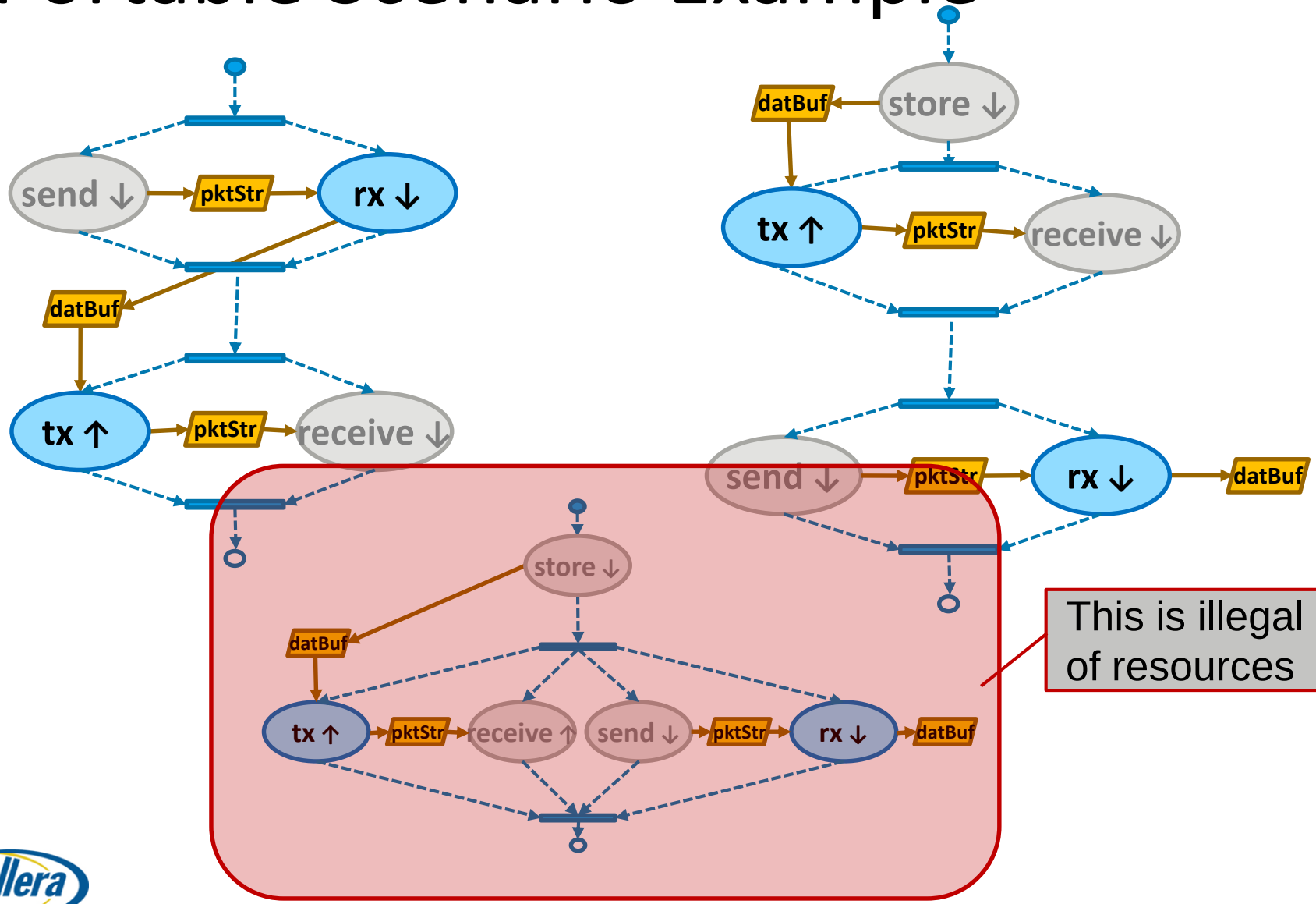

```
component lte_c {
  import data_flow_pkg::*;
  import modem_funcs::*;

  resource mchan_r {…/* struct */};
  pool[1] mchan_r mchan_p;
  bind mchan_p *;

  action rx {
    input pktStr pStr;
    output datBuf dBuf;
    lock mchan_r mchan;
    constraint {…}
  }
  …
}
```

```
action tx {
    input datBuf dBuf;
    output pktStr dBuf;
    lock mchan_r mchan;
    constraint {…}
}
```

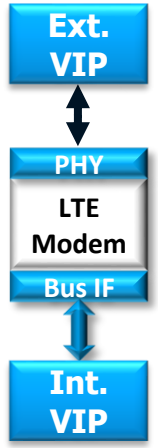


Portable Scenario Example

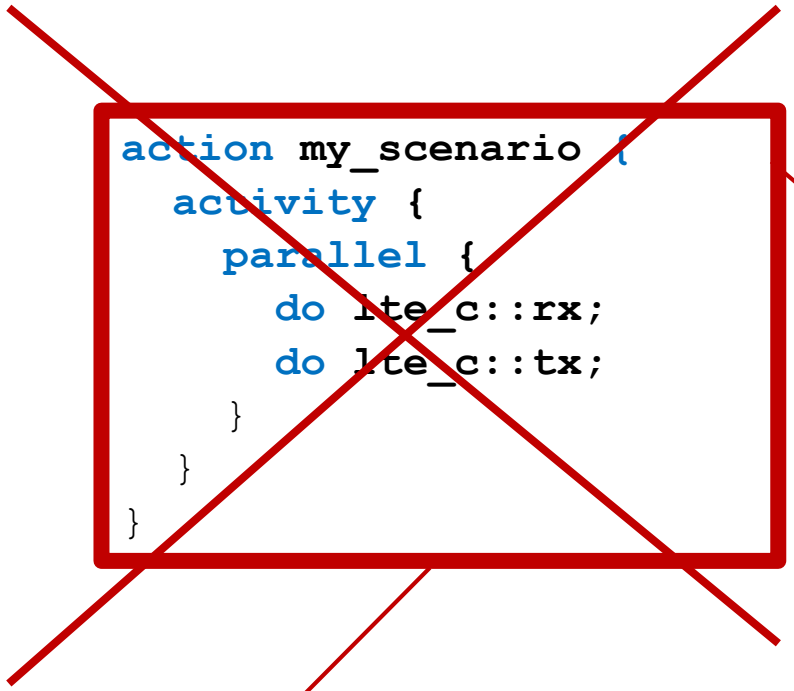
```
action my_scenario {
  activity {
    schedule {
      do lte_c::rx;
      do lte_c::tx;
    }
  }
}
```



This is illegal due to lack of resources



Parallel Scenario



```
action my_scenario {
  activity {
    parallel {
      do lte_c::rx;
      do lte_c::tx;
    }
  }
}
```

Solve time error message

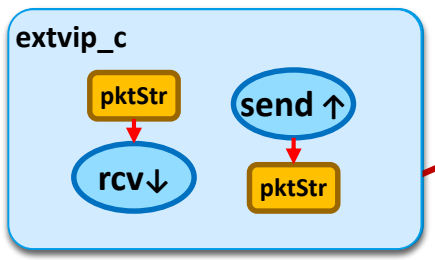
PSS saves you a lot of debug by identifying incorrect scenario early before execution

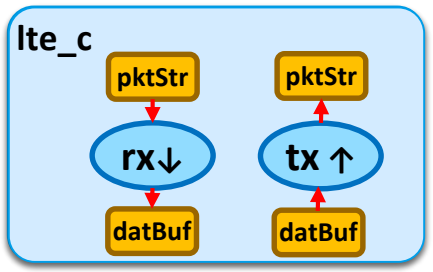
Upfront guarantees with respect to resource, configuration and more

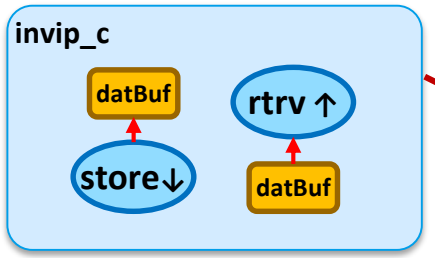
A Block-to-System Example

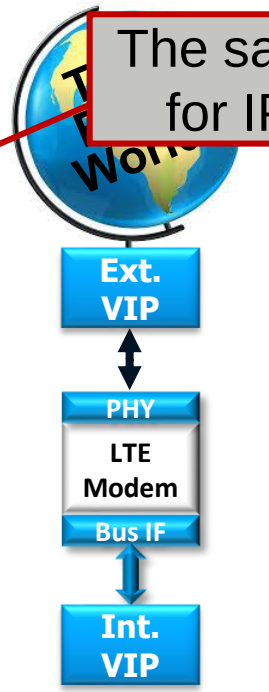
The key thing is that the scenarios are portable

The same actions that are defined for IP verification are reusable

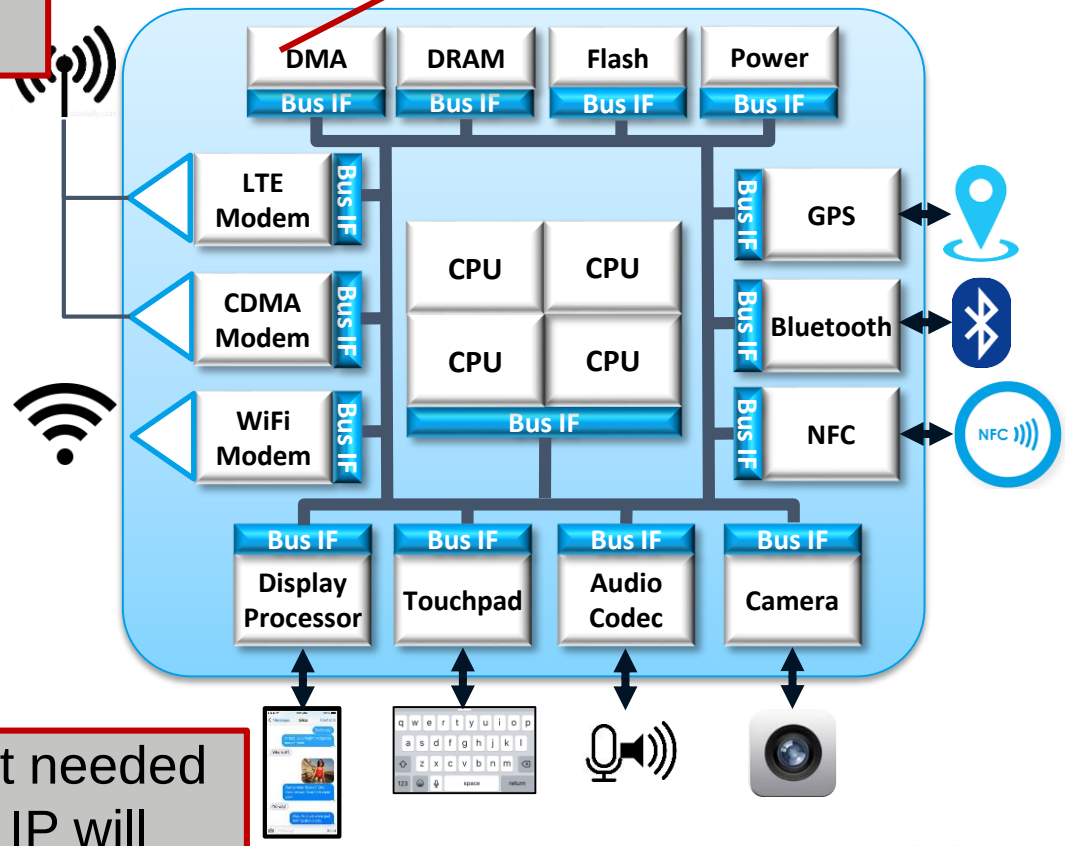








Block

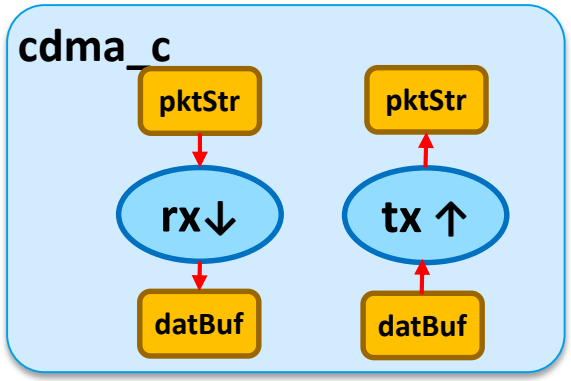


System

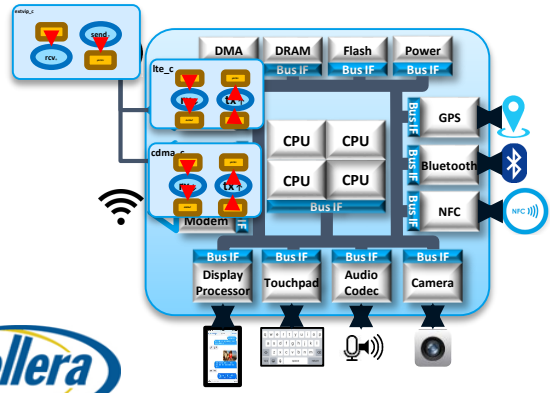
The internal VIP is not needed anymore but a real IP will provide the needed input

The CDMA Modem Component: more of the same …

more **function** imports



post-solve **exec** block runs after randomization



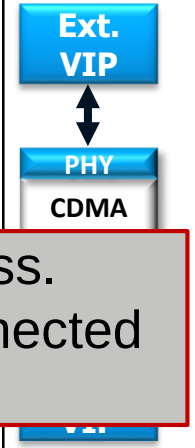

```
package modem_funcs {
  function bit [47:0] CDMA_MAC_src();
  function bit [47:0] CDMA_MAC_dst();
  function bit  [31:0] CDMA_data_buf();
}
component cdma_c {
  import data_flow_pkg::*;
  import modem_funcs::*;

  action rx {
    input pktStr pStr;
    output datBuf dBuf;
    constraint {pStr.size <
```

```
    exec post_solve {
      pStr.addr = CDMA_data_buf();
    }
```

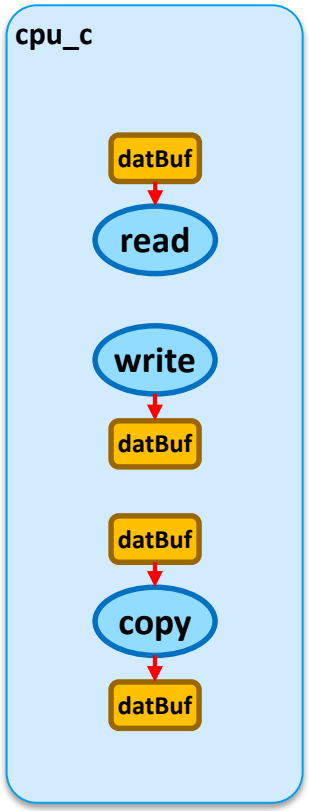
…

```
extend component pss_top {
  ...
  cdma_c cdma;
};
```

PSS Connectivity is seamless.
This component is now connected to all other sub-systems



SW Operations Modeling



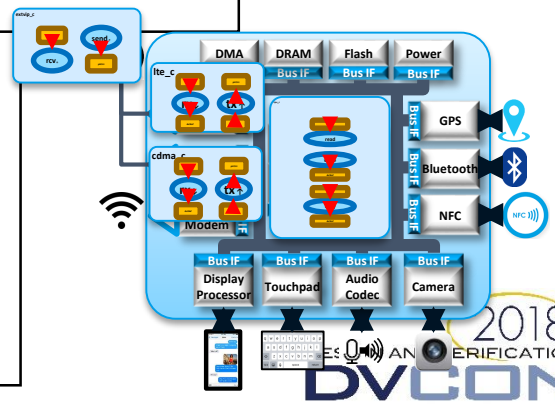

```
component cpu_c {
  abstract action sw_operation {
      lock core_s core;
  };

  action mem_read : sw_operation {
     input datBuf src_data;
  };

  action mem_write : sw_operation
    output datBuf dst_data;
  };

  action mem_copy : sw_operation {
    input datBuf src;
    output datBuf dst;
    constraint c1 {src.size == dst.size;}
    constraint c2 {src.kind == dst.kind;}
  };
};
```

every operation locks the core

For multi-threading PSS also allows resources to be shared

```
component pss_top {
  ...

  pool [4] core_s chan;
  bind core_s *;

};
```

PSS Connectivity done!



The Display Component

```
class screen : public resource {…};


class display_c : public component {
  PSS_CTOR(display_c, component);

  class play : public actio
    PSS_CTOR(play, action);
    input <> data{"data"};
    constraint c {data->kind == video};
    lock <screen> lk {"lk"};
  …};
  type_decl<play> play_d;
};
type_decl<display_c> display_d;
```

C++ input format provides the same constructs with the exact same powerful semantic

declare randomizable enum

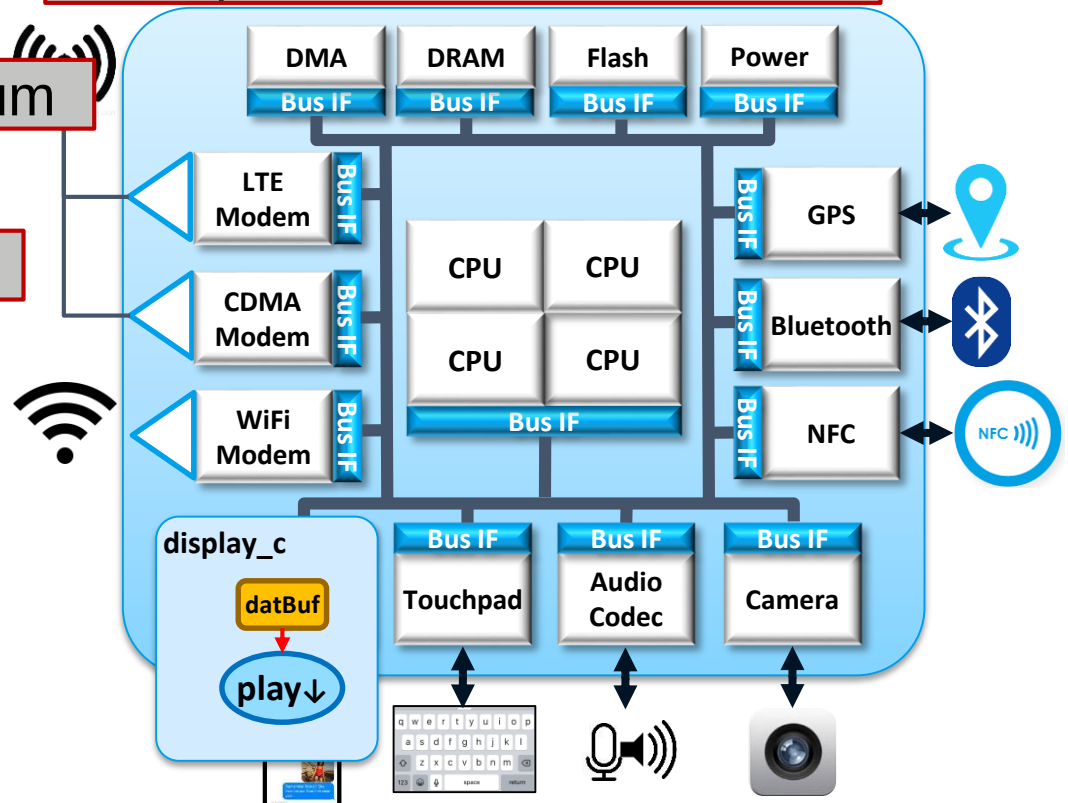
constructor macro

random attribute

constraint

declare type

Note that screen can only accept a video



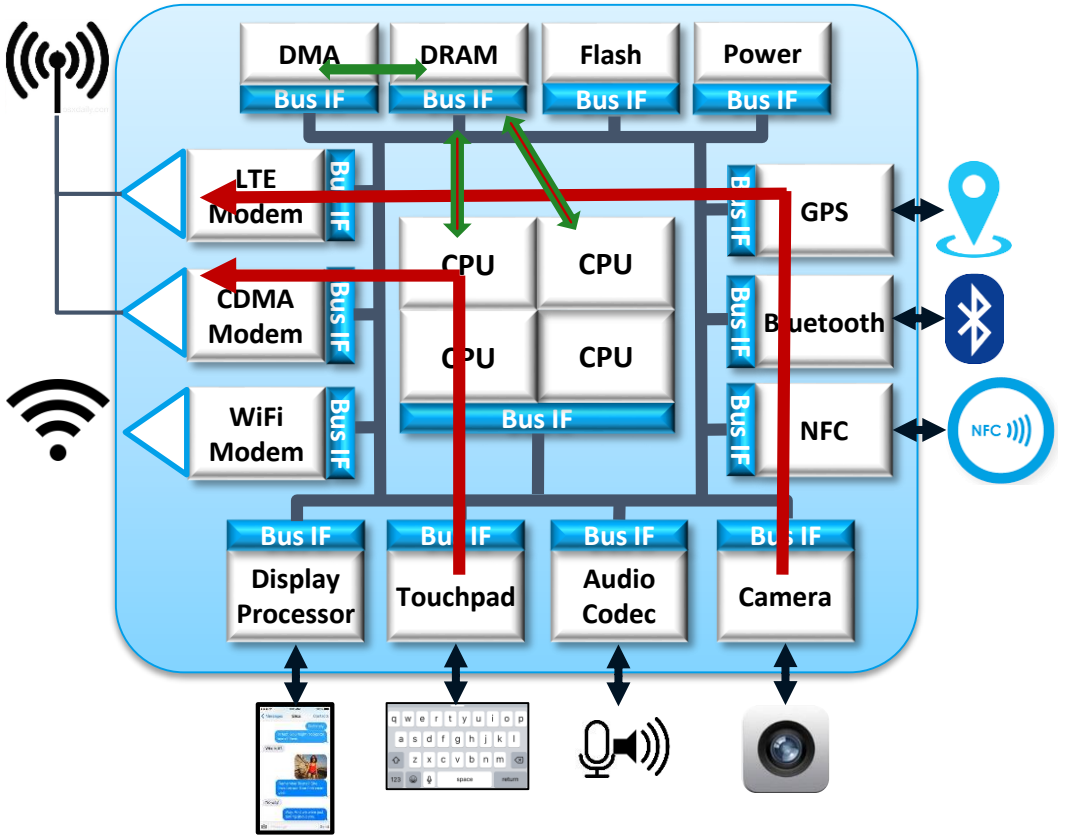
ystem

Example #1: Text Message with Photo

Test requirement: in parallel

1. **Capture a video and upload via the modem**
2. **Send a message to from the keyboard**
3. **Stress the interconnect/DDR with traffic**

```
action my_example1 {
  activity {
      parallel {
          sequence {
              do capture;
              do tx;
              bind capture.dbuf tx.dbuf;
          }
          do txt_msg;
          repeat (10) {
              schedule {
                  do dma_c::xfer;
                  do dma_c::xfer;
                  do cpu_c::mem_copy;
              }
          }
  }}
```

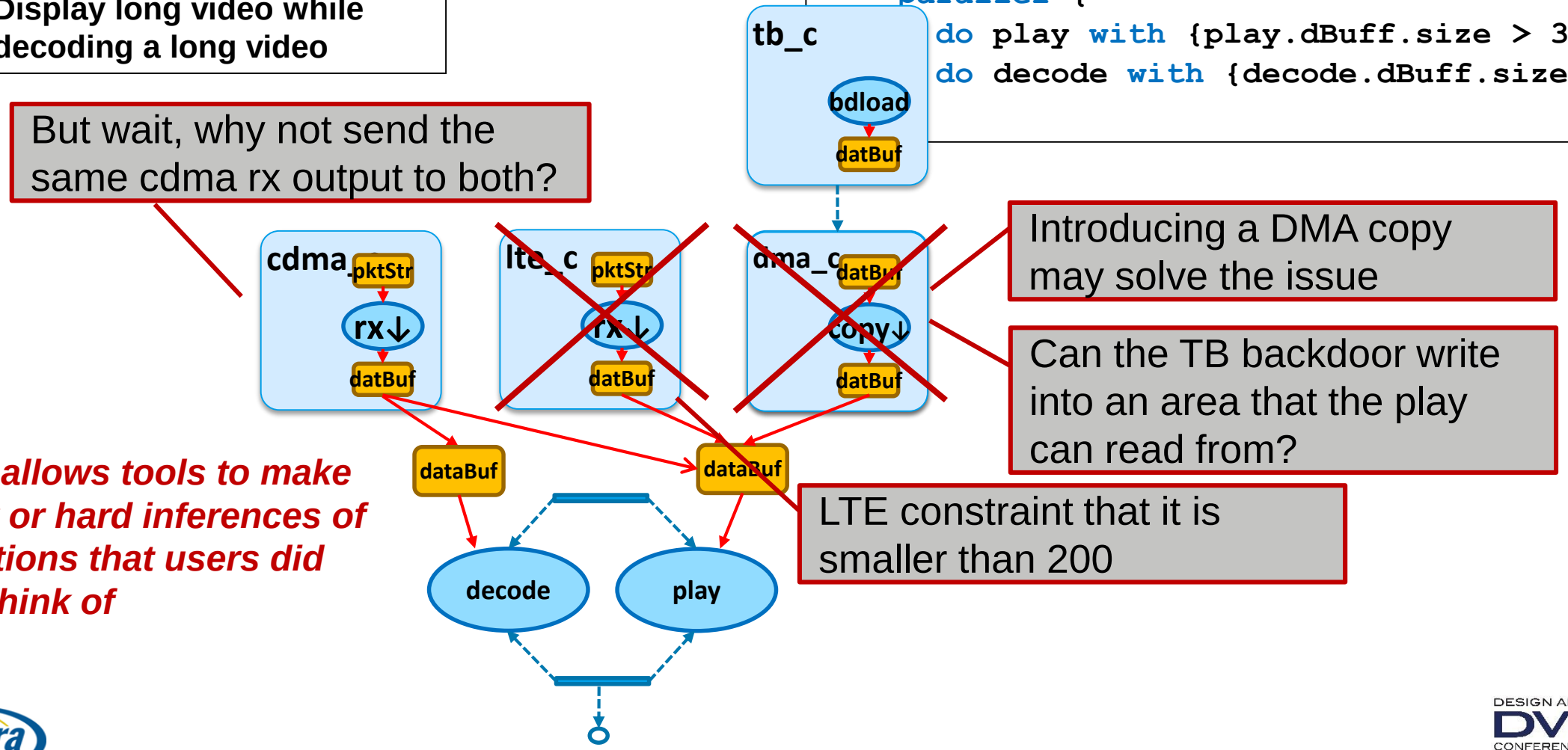


System

Example #2: Partially Specified Scenario

Test requirement:
1. **Display long video while decoding a long video**

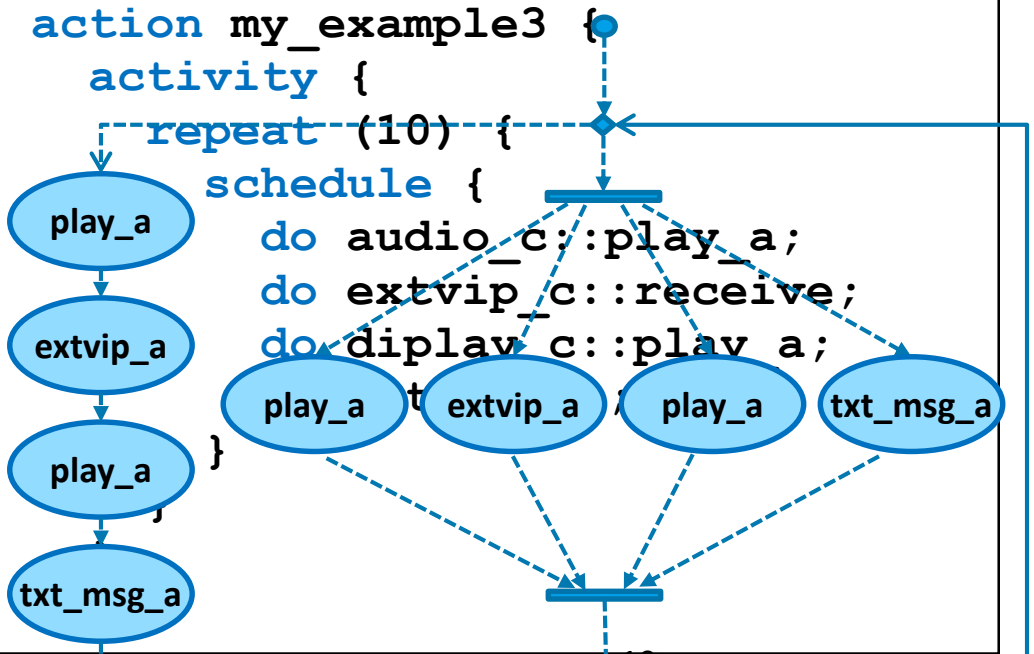
```
action my_example2 {
  activity {
    parallel {
      do play with {play.dBuff.size > 300;}
      do decode with {decode.dBuff.size > 300;}
```



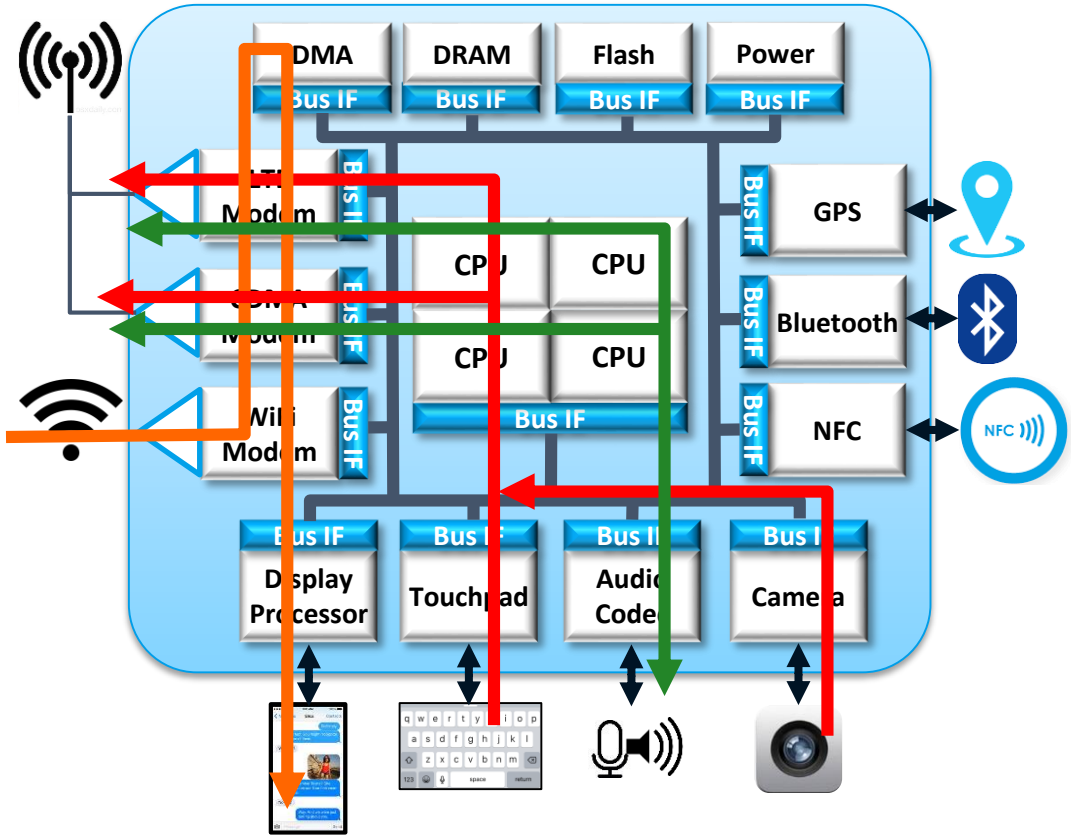
PSS allows tools to make easy or hard inferences of solutions that users did not think of

Example #3

```
action my_example3 {
  activity {
    repeat (10) {
      schedule {
        do audio_c::play_a;
        do extvip_c::receive;
        do diplay_c::play_a;
        t
      }
```



or any parallel or serial combination



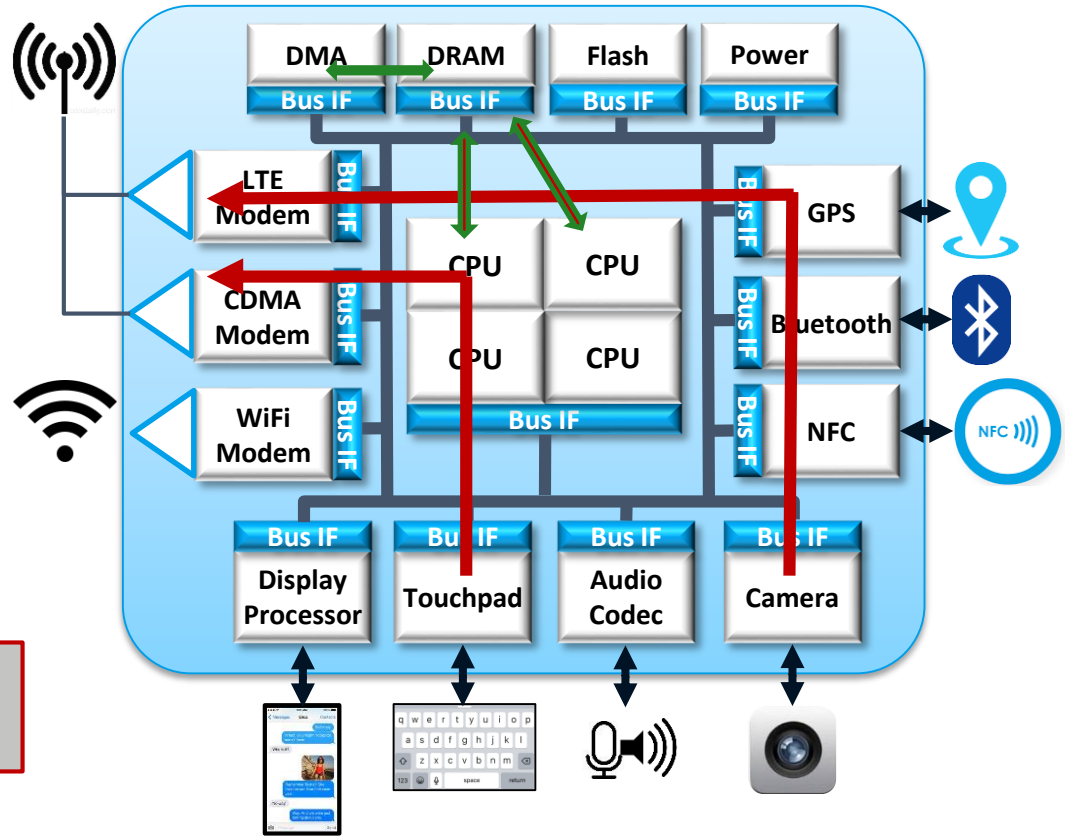
System

Remember This Scenario?

This is our sub-system/IP scenario that can run on full system!

```
action my_scenario {
  activity {
    schedule {
      do lte_c::rx;
      do lte_c::tx;
    }
  }
}
```

We can realize/implement the same scenario on full system



System

Layering in Power Scenarios

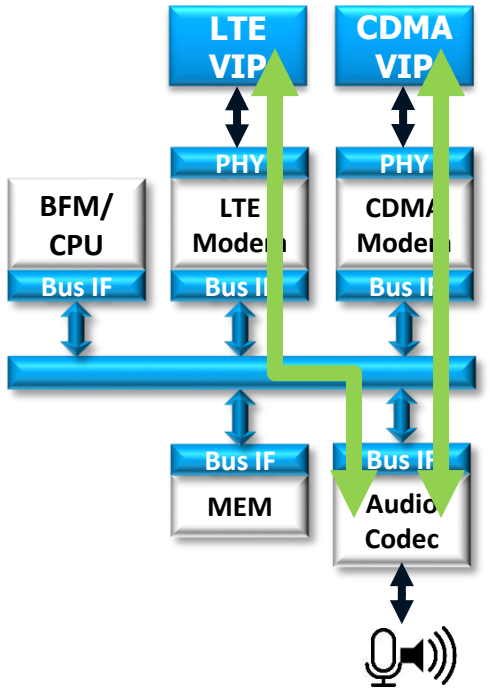
outputs a **radioState** flow object

may only run if previous **rstate** was **off**

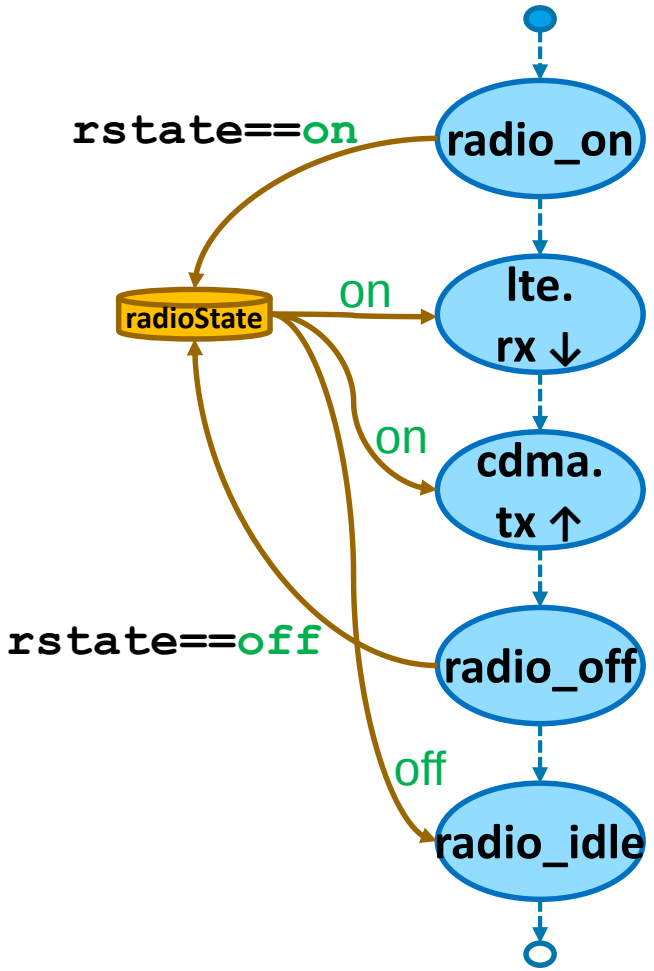
set next **rstate** to **on**

turn on the radio

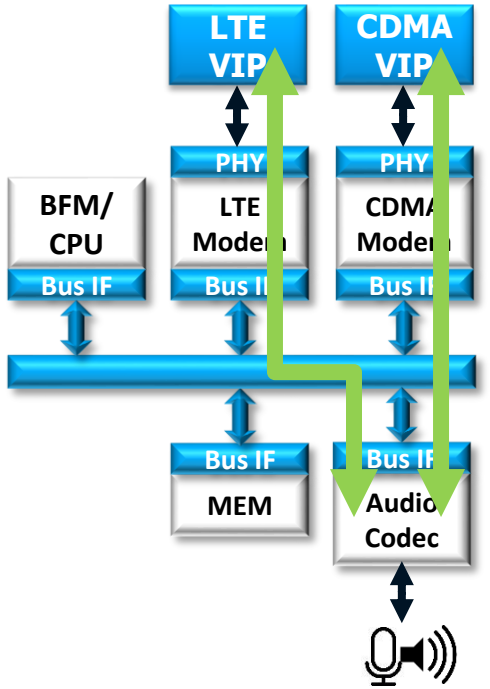
```
extend component pss_top {

  action radio_off {
    output radioState out_s;

    constraint
      out_s.prev.rstate == on;
    constraint
      out_s.rstate == off;

    exec body {
      radio_off();
    }
  }
}
```



Layering in Power Scenarios




```
extend component pss_top {

  action radio_idle {
    input radioState in_s;
    constraint in_s.rstate == off;
  }

  action test {
    activity {
      select {
        do radio_idle;
        schedule {
          do audio_c::play;
          do lte_c::tx;
        }
      }
    }
  }
}
```



Portable Stimulus Coverage

- PSS provides full functional coverage on top of the concise behavioral model
  - Allows capturing coverage goals that are impractical in any other way
  - Due to the declarative nature of PSS the coverage goal can direct the randomization
- Coverage constructs style derived from SV
  - Support cross, illegal, ignore and others
  - Keyword is change from covergroup -> coverspec
- Supported coverage
  - Data structure coverage
  - Action coverage
  - Scenario (compound action) coverage
  - Resource coverage

Actions model captures the infinite legal scenario space

==

Coverage capture the verification intent

Coverage Example

```
action my_example1 {
  dma_C::xfer xfer1;
  dma_C::xfer xfer2;
  camera_c::capture capture1;

  activity {
      parallel {
         sequence {
            capture1;
            do tx;
            bind capture.dbuf tx.dbuf;
         }
         do txt_msg;
         repeat(10){
            schedule {
               xfer1;
               xfer2;
               do cpu_c::copy;
         }  }
  }}
```

```
extend action my_example1 {
   coverspec my_example1_goals
                        (my_example1 t) {
      xfer1_chan: coverpoint
            t.xfer1.channel.instance_id;
      xfer2_chan: coverpoint
            t.xfer2.channel.instance_id;
      chan_cross: cross xfer1_chan,
                              xfer2_chan;
      video_format: coverpoint
          t.capture1.dBuf.video_format;
   }
}
```

Advantages of Portable Stimulus

- Declarative language/library enables automation
  - Partial specification expands to multiple scenarios
  - Constrained-randomization at the scenario level
- Easily composable, reusable and portable
- Maps to implementations on multiple platforms
  - Leverage existing infrastructure, sequences, methods
  - Apply scenario randomization and coverage to system-level tests (in C, too!)

Moving Forward

- Established Communications
  - Main PSWG Site **http://www.accellera.org/community/portable-stimulus**
    - Central site to access standards, draft and tutorial for PSS
  - Public forum for 1.0 feedback

    http://forums.accellera.org/forum/44-portable-stimulus-10
    - Please review the standard and provide feedback. We truly appreciate your feedback!
- 1.1 Activities
  - 1.0 Errata release planned
  - PSWG pioneering requirements management tool deployment at Accellera
  - Moving to rigorous tracking based on requirements (not only Mantis)