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

Accellera Portable Stimulus Working Group
It's an SOC World

Number of Embedded Processor Cores

72% of ASIC designs contain embedded processors
59% of FPGA designs contain embedded processors

Block-to-System Test Reuse

- UVM constrained-random is great for block-level testing
  - UVM Sequences model both external & internal behavior
  - Block tested standalone
- SOC-level tests driven by use-cases
  - Embedded SW drives the test
  - Usually written in C
- Still want to test the block
  - Need to reuse test intent
  - Coordinate with other traffic

Accellera PSWG
Reuse of Test Intent Across Platforms/Users

- Single specification of test intent is critical
- Define "scenario space" by capturing:
  - interactions
  - dependencies
  - resource contention
- Abstraction lets tool automate generation
  - Multiple targets
  - Target-specific customization
Biggest Verification Challenges

Creating Sufficient Tests to Verify the Design (Coverage Closure)

Knowing my Verification Coverage

Managing the Verification Process

Time to Isolate & Resolve Bug

Time to Discover Next Bug

Defining Appropriate Coverage Metrics

Other


Accellera PSWG
Maximize Productivity by Separating Concerns

Directed:
start(Boston);
drive(West, I90, Buffalo);
if(!Canada)
drive(West, I90, Chicago);
drive(West, I80, Salt Lake City);
drive(West, I84, Portland);
else
drive(North, I190, Niagara Falls);
drive(West, ON403, Flint);
drive(North, I75, Fargo);
drive(West, I94, Billings);
drive(West, I90, Ritzville);
drive(South, US395, Stanfield);
drive(West I84, Portland);

Accellera PSWG
Maximize Productivity by Separating Concerns

Constrained-Random:

class drive;
  rand city_e start, end;
  rand dir_e direction;
endclass

class directions;
  task body;
    drive.go() with {start == Boston, dir == West;}
    while(drive.end != Portland)
      drive.go();
endtask

constraint NoCanada{start==Buffalo -> dir == West;}
constraint Chi {start==Chicago -> dir == West;}
...
endclass

class drop_friend extends directions;
  constraint Chi {start==Chicago -> dir inside [North, West];
  start==Chicago -> end==Minneapolis;}
endclass

class sightsee extends drop_friend;
  constraint DT {start==Minneapolis -> end==DevilsTower;}
  constraint MR {start==DevilsTower -> end==MtRushmore;}
  constraint MT {start==MtRushmore -> end==Billings;}
endclass

Testwriter must still manage details
Global Optimization is Difficult
Maximize Productivity by Separating Concerns

Declarative:
start == Boston;
end == Portland;

set_point(Minneapolis);
set_point(MtRushmore);
set_point(DevilsTower);
set_preference(West);
...

Testwriter focuses on INTENT Tools handle the details
Modeling Portable Stimulus Requires Abstraction

• Begin with the end in mind
  – Translating one language into another is hard
  – Each target language has its own semantics

• Abstraction lets us focus on common semantics
  – Schedule well-defined behaviors
  – Scheduling semantics allow scenario exploration

• Single partial specification expanded into multiple scenarios
So, How Is This All Going to Work?

- **Abstract Portable Stimulus Model**
  - High-level *test intent*

- **PSS Processing Tool**
  - Builds the semantic data model
  - Assembles the static structure

- **PSS Solving Platform**
  - Solves scheduling and algebraic relationships
  - "Gen time" to create executable code
  - "Run time" for simulation

- **Target-Platform Scenario Implementation**
  - Executable solution of the partial specification

- **Target-Platform User Code**
  - User code seamlessly integrated

---

Accellera PSWG
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
Projected Tool Flow

**DSL**

- **Action per top**
  - `ad_0` by
  - `ad_1` by
  - `ad_2` by
- **Activity**
  - `by`

**C++**

```cpp
class gen_top : public action {
public:
    gen_top(uvm_module* module)
    : action(module)
    {
        activity = {
            by
        };
    }
};
```

**Scenario model + Constraints**

**Compile/Link**

**Run**

**Constraint Solver**

**Solved Model**

**Test Generator**

- Gen-time or Run-time

**Abstract Partial Specification**

- **Accellera PSWG**

---

**UVM**

- `int main(int argc, char **argv) {
  // ...
  return 0;
```
Hello, World
Hello World: Atomic Actions

- **Component** groups elements for reuse and composition
- **Action** defines behavior
- **Exec** defines implementation

```verbatim
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
Hello World: Compound Actions

```c
component pss_top {
  action hello_a {
    exec body C = """
    printf("Hello\n");
    """;
  }
  action world_a {
    exec body C = """
    printf("World\n");
    """;
  }
}
```
Hello World: Compound Actions

compound action traverses other actions
activity defines scheduling

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

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

✓ Behavior encapsulation
✓ Behavior scheduling
Hello World: Data Flow Objects

```verilog
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

```plaintext
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
```

Accellera PSWG
Hello World: Packages

```
package hw_pckg {  
  buffer msg_buf {  
    rand string s;  
  }  
}
```

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

```
on send_a {  
  output msg_buf msg;  
}
```

```
on hello_world_a {  
  end_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;  
  }
```

✅ 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

```verilog
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";}
        }
    }
}
```
Activity: Robust Expression of Critical Intent

```action
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

```pss
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;
  }
}
```

- **Type extension**
- **Inheritance**
- **Override**

**Type extension**
- **Object-oriented inheritance**
- **Type (& instance) override**
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

Multiple scenarios from simple specification

```action
action hello_world_a {
  activity {
    sequence {
      do disp_h;
      do disp_w;
    }
  }
}
```
Hello World: C++

```cpp
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++

```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 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++

```cpp
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;
```
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
Block-to-System Example
A Block-to-System Example
Define Actions

• What does the Modem do?
  – Receive packet: rx
  – Transmit packet: tx

• What data flow objects does the Modem use?
  – External Interface: packet
  – Internal Interface: datStr

• What does the External IP do?
  – send packets
  – receive packets

• What does the Internal IP do?
  – Store datStrs
  – Retrieve datStrs
Define Data Flow Objects

```
package data_flow_pkg{
    enum dir_e {inb=0, outb};
    stream datStr {
        rand dir_e dir;
        rand bit [7:0] length;
        rand bit [31:0] addr;
    }
    stream packet {
        rand dir_e dir;
        rand bit [15:0] size;
        bit [47:0] MAC_src;
        bit [47:0] MAC_dst;
    }
}
```

**enum** defines a set of integral named constants

**stream** requires *parallel* producer-consumer execution

**rand** fields are randomized
package modem_funcs {
    import data_flow_pkg::dir_e;
    function void set_mode(dir_e dir);
}

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

    action rx_a {
        input packet pkt;
        output datStr bPkt;
        constraint {pkt.dir == inb; bPkt.dir == inb};
        exec body {
            set_mode(pkt.dir);
        }
    }

    action tx_a {
        input datStr bPkt;
        output packet pkt;
        constraint {pkt.dir == outb; bPkt.dir == outb};
        exec body {
            set_mode(pkt.dir);
        }
    }
    ...
}
The VIP Components

**extvip_c**
- **send** →
- **receive** ↑

```
component extvip_c {
  import data_flow_pkg::packet;
  action send_a {
    output packet pkt;
    constraint {pkt.dir == inb;}
  }
  action receive_a {
    input packet pkt;
    constraint {pkt.dir == outb;}
  }
}
```

**intvip_c**
- **store** ↓
- **rtrv** ↑

```
component intvip_c {
  import data_flow_pkg::datStr;
  action store_a {
    input datStr pkt;
    constraint {pkt.dir == inb;}
  }
  action rtrv_a {
    output datStr pkt;
    constraint {pkt.dir == outb;}
  }
}
```
Putting it together

```typescript
component pss_top {
    import data_flow_pkg::*;
    extvip_c xvip;
    lte_c lte;
    intvip_c ivip;

    pool packet pkt_p;
    bind pkt_p {xvip.*, lte.*};
    pool datStr bPkt_p;
    bind bPkt_p {ivip.*, lte.*};

    action test {
        activity {
            schedule {
                do lte_c::rx_a;
                do lte_c::tx_a;
            }
        }
    }
}
```
Putting it together

activity {
    schedule {
        do lte_c::rx_a;
        do lte_c::tx_a;
    }
}
Resources: Target-Specific Constraints

• What if the Modem is half-duplex?
  – Prevent rx & tx from running in parallel

• PSS models target-specific resources
  – May be assigned to an action for its duration
  – Exclusive (locked) or non-exclusive (shared)
The Modem Component + Resources

```vhdl
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_a {
    input packet pkt;
    output datStr bPkt;
    lock mchan_r mchan;
    constraint {pkt.dir == inb; bPkt.dir == inb;}
  }
  ...
}
```

- `resource` defines a resource object
- `pool` defaults to `size == 1`
- `lock` declares exclusive access
Claiming Resource Objects

• Actions may lock or share resources
  – Actions that lock a given resource may not overlap
  – Actions that share a given resource may overlap

```c
resource chan_r {...};
pool [2] chan_r chan_p;
bind chan_p {*};
action rlk_a {
  lock chan_r chan;
  ...};
action rsh_a {
  share chan_r chan;
  ...};
```

• A total of size locking actions may execute in parallel for a given resource pool
Solution Space Mapping

```
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;
}
}
```

```
buffer mbuf {...};

action do_a {
  output mbuf m;
...
}

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

action do_c {
  output mbuf m;
...
}

action do_d {
  input mbuf o;
  output mbuf o;
...
}

action do_g {
  output mbuf m;
...
}

action do_j {
  output mbuf m;
...
}
```

Partial Specifications are Flexible
Resolving a Scenario

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

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

Are there any resource conflicts that constrain the possible scheduling?
A Block-to-System Example
Modem Sub-system

- Manage Voice Calls
- Can use either
  - LTE Modem, OR
  - CDMA Modem
- Both modems exchange common datStr data with Audio Codec
- *Stream* relationship between Modem and Audio Codec:
  - both must operate concurrently
Exec Block Types
Specify mapping of PSS entities to their implementation

pre_solve
constraint solving
post_solve
body
pre_solve
constraint solving
post_solve
body
run_start

run_end

documentation of PSS entities and their implementation

#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() {
    
}

gcc -c test.c -DBARE_METAL
test.sh

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

Could be SV or other language

Accellera PSWG
The CDMA Modem Component

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_a {
        input packet pkt;
        output datStr bPkt;
        constraint {pkt.dir == inb; bPkt.dir == inb;}

        exec pre_solve {
            pkt.MAC_src = CDMA_MAC_src();
            pkt.MAC_dst = CDMA_MAC_dst();
        }
    }
}

stream packet {
    rand dir_e dir;
    rand bit [15:0] size;
    bit [47:0] MAC_src;
    bit [47:0] MAC_dst;
}
The CDMA Modem Component

```vhdl
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_a {
        input packet pkt;
        output datStr bPkt;
        constraint {pkt.dir == inb; bPkt.dir == inb;}
    }

    stream datStr {
        rand dir_e dir;
        rand bit [7:0] length;
        rand bit [31:0] addr;
    }

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

more function imports

random variable

post-solve exec block runs after randomization

cdma_c

rx ↓

tx ↑
The Audio Codec Component

```
package audio_funcs {
    function void play(bit[31:0] addr, bit[7:0] len);
    function void record(bit[31:0] addr);
}
component audio_c {
    import audio_funcs::*;

    action rec_a {
        output datStr bPkt;
        constraint {bPkt.dir == outb;
                     bPkt.length == 1024; }

        exec body {
            record(bPkt.addr);
        }
    }
}
```

Accellera PSWG
Putting it together

```component pss_top {
  import data_flow_pkg::*;
  extvip_c lte_vip, cdma_vip;
  lte_c lte;
  cdma_c cdma;
  audio_c audio;

  action test {
    activity {
      schedule {
        do audio_c::play_a;
        do extvip_c::receive_a;
      }
    }
  }
}
```
activity {
    schedule {
        do audio_c::play_a;
        do extvip_c::receive_a;
    }
}
Layering in Power Scenarios

package power_state {
    function void radio_on();
    function void radio_off();

    enum radio_state_e { on, off };

    state radioState {
        rand radio_state_e rstate;
        constraint
        initial -> rstate == off;
        ...
    }

    extend component pss_top {
        pool radioState radio_p;
        bind radio_p *;
    }
}
Layering in Power Scenarios

package power_state {

    extend action lte_c::rx_a {
        input radioState in_s;
        constraint
            in_s.rstate == on;
    }

    extend action cdma_c::rx_a {
        input radioState in_s;
        constraint
            in_s.rstate == on;
    }

    ...}

Subsystem
package power_state {
    
    extend action lte_c::tx_a {
        input radioState in_s;
        constraint
        in_s.rstate == on;
    }
    
    extend action cdma_c::tx_a {
        input radioState in_s;
        constraint
        in_s.rstate == on;
    }
    
    ...
}

Subsystem
Layering in Power Scenarios

```java
extend component pss_top {

    action radio_on_a {
        output radioState out_s;

        constraint
            out_s.prev.rstate == off;
        constraint
            out_s.rstate == on;
        exec body {
            radio_on();
        }
    }
}
```

outputs a `radioState` flow object

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

set next `rstate` to **on**

turn on the radio

Subsystem
Layering in Power Scenarios

```plaintext
extend component pss_top {
  action radio_off_a {
    output radioState out_s;
    constraint
      out_s.prev.rstate == on;
    constraint
      out_s.rstate == off;
    exec_body {
      radio_off();
    }
  }
}
```

outputs a radioState flow object
may only run if previous rstate was on
set next rstate to off
turn off the radio

Subsystem
Layering in Power Scenarios

```
extend component pss_top {
    action radio_idle_a {
        input radioState in_s;
        constraint in_s.rstate == off;
    }

    action test {
        activity {
            select {
                do radio_idle_a;
                schedule {
                    do audio_c::play_a;
                    do extvip_c::receive;
                }
            }
        }
    }
}
```
UVM + C-Test Tool Flow

Subsystem

Accellera PSWG
Portable Stimulus Coverage

• Coverage constructs derived from SV
  – Support cross, illegal, ignore and others
  – Keyword is change from covergroup -> coverspec

• Coverage is currently data-centric
  – Monitor values and ranges on action/struct fields

• More coverage types may be added
  – Action Coverage
  – Scenario (Action Sequence) Coverage
  – Datapath Coverage
  – Resource Coverage
Coverage

```cpp
action setup_modem {
    enum direction_e {TRANSMIT, RECEIVE, BOTH};
    rand direction_e direction;
    unsigned int baud_rate;
    unsigned int packet_size;
    unsigned bit [1:0] destination_addr;

    exec body {.... }

coverspec modem_initialization (init_modem) {
    constraint baud_len_c {
        if (direction == TRANSMIT) {
            baud_rate in [28000,3192704, 4196704];
        }
    }
    baud: coverpoint init_modem.packet_size {
        bins size [28000 ... 4296704]/32;
    }
    dir : coverpoint transmit_dir_tx {
        bins transmit = {TRANSMIT};
        bins receive = {RECEIVE};
        bins bidi = {BOTH};
    }
    transmit_type_invld : cross transmit_dir_tx, addr {
        ignore addr ? (direction == TRANSMIT) : 1;
    }
    address: coverpoint addr ;
}
```
Monitoring Coverage

• Stimulus monitoring
  – Generation time tool can output what it generated/scheduled
    • As long as test “passes”, the coverage data is valid

• Runtime state monitoring
  – Requires generation of monitoring code
    • May be C/C++ code running on target cpu
      – e.g. data sent out “trickbox” mechanism
    • May be “off-chip” monitoring via test ports or other communication ports
A Block-to-System Example

Block

Subsystem

System
Full System Scenarios

- Reuse Sub-System Voice Call model
- Add Streaming Video over Wifi
- Add Text Message with Photo
Streaming Video over Wifi

System
Streaming Video over Wifi

```cpp
PSS_ENUM (kind_e, video, photo);
class datBuf : public buffer {
    PSS_CTOR (datBuf, buffer);
    rand_attr<kind_e> kind {"kind"};
...};

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

class play_a : public action {
    PSS_CTOR (play_a, action);
    input <datBuf> data {"data"};
    constraint c {data->kind == video};
    lock <screen> lk {"lk"};
...
    type_decl<play_a> play_d;
};
type_decl<display_c> display_d;
```
Streaming Video over Wifi

class screen : public resource { … };

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

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

type_decl<display_c> display_d;
Text Message with Photo
component touchpad_c {
    action enter_a {
        output message msg;
        lock screen lk;
        ...
    }
}
component pss_top {

  action txt_msg_a {
    camera_c::capture_a capture;
    touchpad_c::enter_a enter
    cpu_c::send_msg send;

    bind capture.out_photo send.in_photo;
    bind enter.out_msg send.in_msg;

    activity {
      capture;
      enter;
      send;
    }
  }
}
Putting it all together
Putting it all together
Putting it all together
component pss_top {
  // imports
  // instantiations
  // pools & binds

  action test {
    activity {
      repeat (10) {
        schedule {
          do audio_c::play_a;
          do extvip_c::receive;
          do dispaly_c::play_a;
          do txt_msg_a;
        }
      }
    }
  }
}
Putting it all together

or any parallel or serial combination
Introducing the Panel

• Faris Khundakjie, Intel, PSWG Chair
• Dave Brownell, Analog Devices, PSWG Secretary
• Sharon Rosenberg, Cadence
• Srivatsa Vasudevan, Synopsys
• Karthick Gururaj, Vayavya
• Tom Fitzpatrick, Mentor
• Adnan Hamid, Breker