

Learning the Portable Stimulus Standard

Tom Fitzpatrick

CEO Big Fish EDA Consulting

On behalf of the Accellera Systems Initiative

and the Accellera Portable Stimulus Working Group



PSS Communication

▶▶▶ Session 06

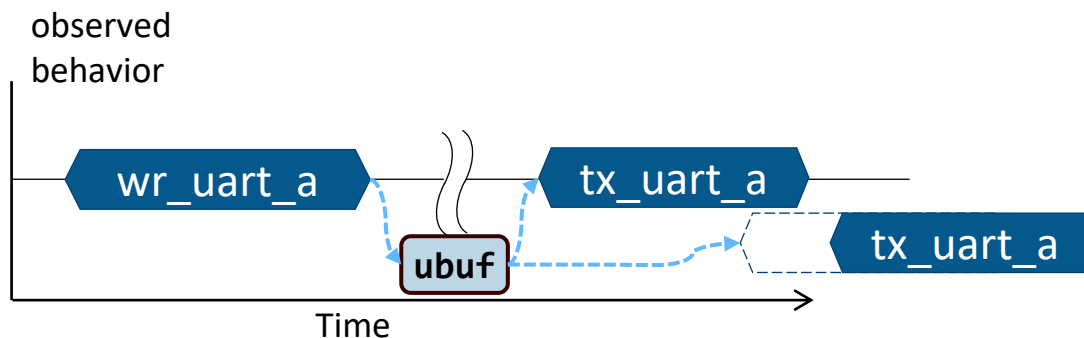
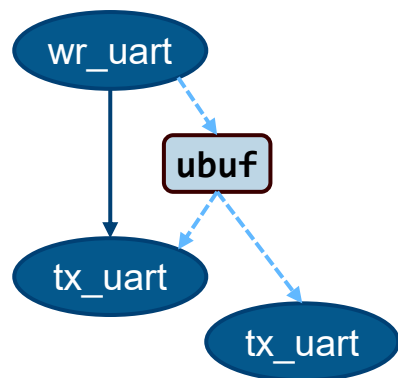
What You Will Learn

- ▶ Modeling data communication in PSS
- ▶ Modeling state and state machines
- ▶ Managing communication using pools
- ▶ Modeling system resources

::: PSS Data Flow: buffer

- ▶ Specialized *struct* type
- ▶ Persistent storage
 - ▶ Can be written and read
 - ▶ Producer must complete before consumer starts
- ▶ One-producer-to-many-consumers

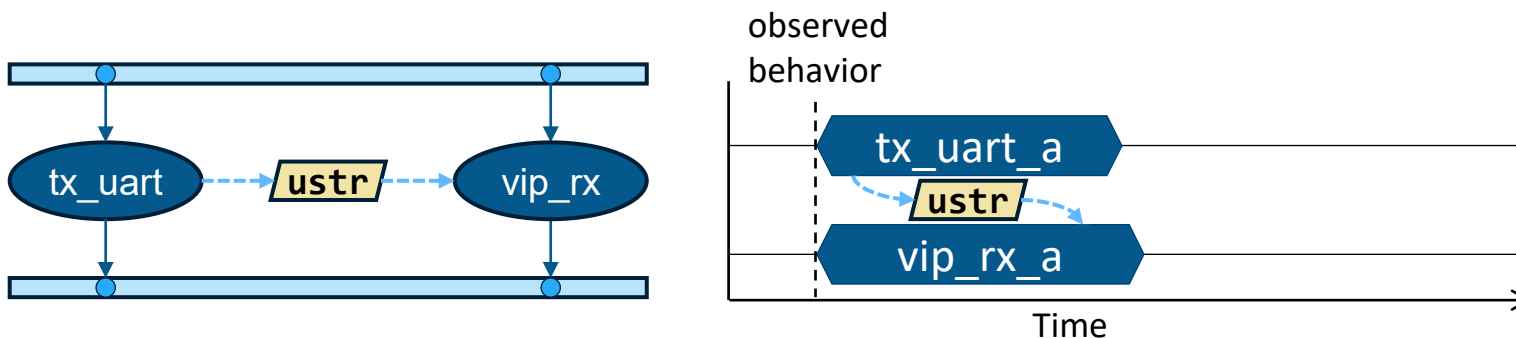
```
buffer ubuf_b {  
    rand bit[8] data;  
    rand bool odd_parity;  
    rand bit parity;  
}  
  
action wr_uart_a {  
    output ubuf_b obuf;  
}  
  
action tx_uart_a {  
    input ubuf_b ibuf;  
}
```



::: PSS Data Flow: stream

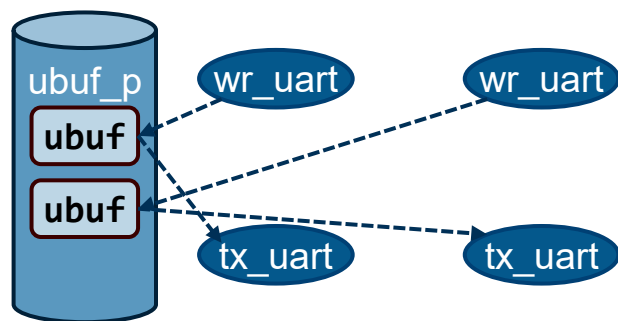
- ▶ Specialized *struct* type
 - ▶ Transient data communication
- ▶ One-producer-to-one-consumer
 - ▶ Producer and consumer traversed in *parallel*

```
stream ustr_s {  
  rand bit[8] data;  
  rand bit parity;  
  rand bit i2o;  
}  
  
action vip_rx_a {  
  input ustr_s istr;  
}  
  
action tx_uart_a {  
  input ubuf_b ibuf;  
  output ustr_s ostr;  
}
```



Flow Objects Managed via Pools

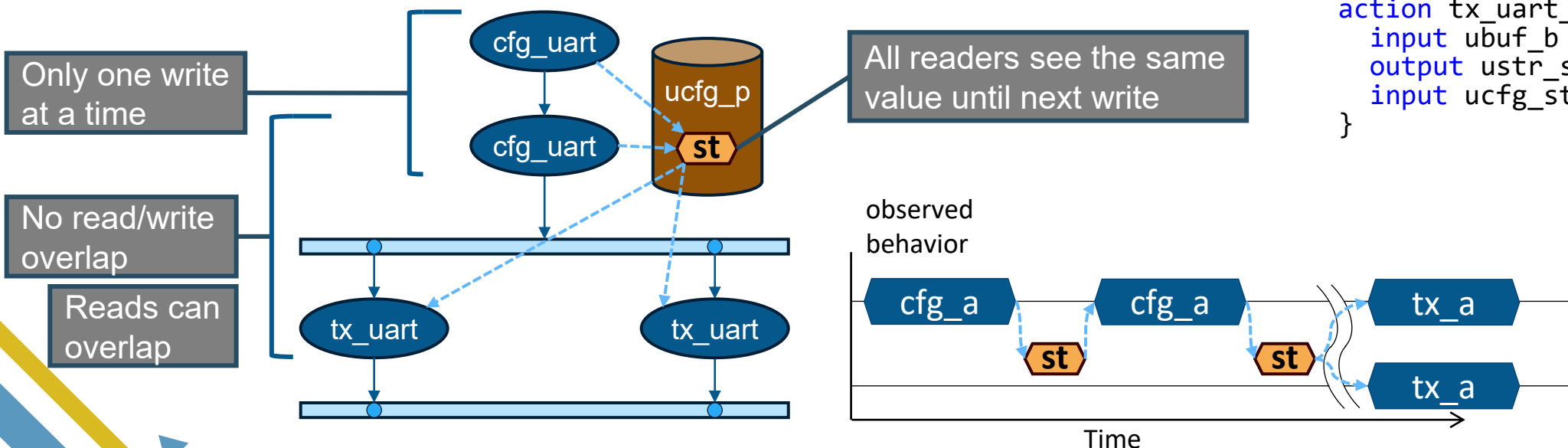
- ▶ A pool is a collection of objects
- ▶ Action ports are bound to pools
- ▶ Actions input/output objects from/to pools
- ▶ Buffer/stream pools contain as many elements as needed
- ▶ More details in the next session



... PSS Data Flow: state

- ▶ Holds the state of an element at a particular time
- ▶ State pools have only one element
- ▶ May have multiple writers and multiple readers
- ▶ Semantics prevent race conditions

```
state ucfg_st {  
  rand bit[8] in [4..128] baud_div;  
  rand bool odd_parity;  
}  
  
action cfg_uart_a {  
  output ucfg_st ocfg;  
}  
  
action tx_uart_a {  
  input ubuf_b ibuf;  
  output ustr_s ostr;  
  input ucfg_st icfg;  
}
```

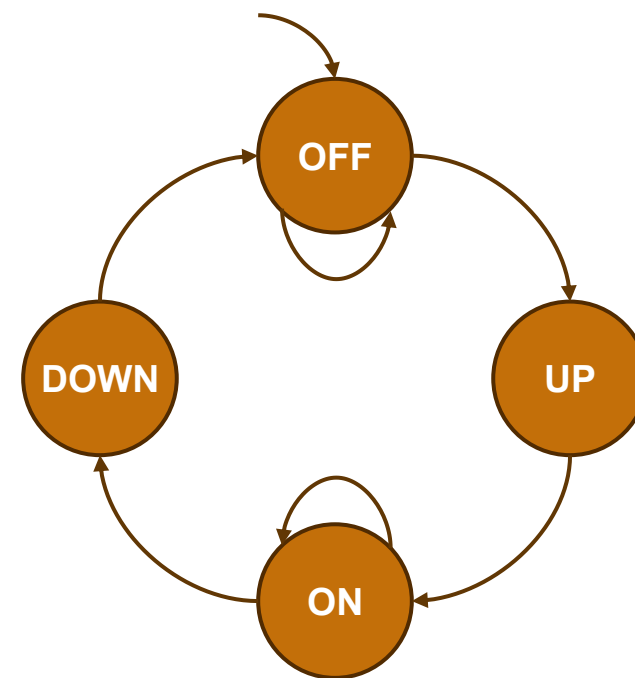


Modeling a Finite State Machine

- ▶ Actions can input and output the same state type
- ▶ Constraints define state transitions

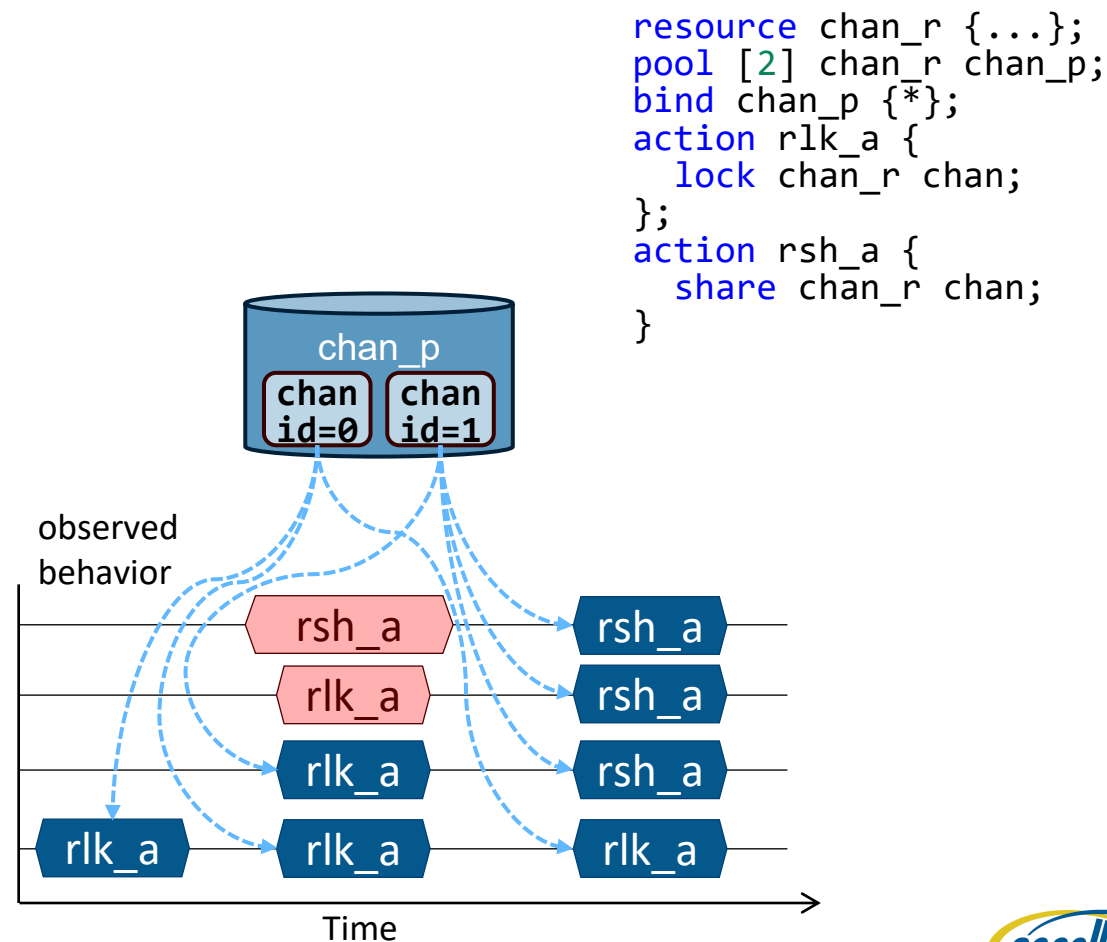
```
enum power_e { OFF, UP, ON, DOWN };
state ucfg_st {
  rand power_e upwr;
  constraint {initial -> upwr == OFF;}
}
action upwr_trans_a {
  input ucfg_st icfg;
  output ucfg_st ocfg;

  constraint transition_c {
    (icfg.upwr == OFF) -> (ocfg.upwr in [UP,OFF]);
    (icfg.upwr == UP) -> (ocfg.upwr == ON);
    (icfg.upwr == ON) -> (ocfg.upwr in [ON,DOWN]);
    (icfg.upwr == DOWN) -> (ocfg.upwr == OFF);
  }
}
```



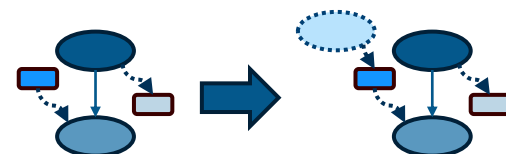
Resource Objects

- ▶ Represent some element of the execution environment
- ▶ Actions **lock** (exclusive) or **share** resources
- ▶ Resource pools hold a discrete number of resources
- ▶ Claimed by an action for its duration
- ▶ Each resource in a pool has a unique **instance_id** field



::: Flow/Resource Objects and Activities

- ▶ Activities define the schedule of action traversals
- ▶ Flow objects add scheduling constraints between actions
 - ▶ Buffer: sequential producer/consumer
 - ▶ Stream: parallel producer/consumer
 - ▶ State: sequential read/write or write/write
 - ▶ Resource: sequential locking of the same resource
- ▶ If the flow/resource constraints cannot be met within the activity, the solver must *infer* additional actions to satisfy them



```
activity {  
  do act1_a;  
  do act2_a;  
}
```



Thank You

