Mentor Graphics®

UVM Connect

Part 1 – Introduction

Adam Erickson
Verification Technologist

academy@mentor.com
www.verificationacademy.com

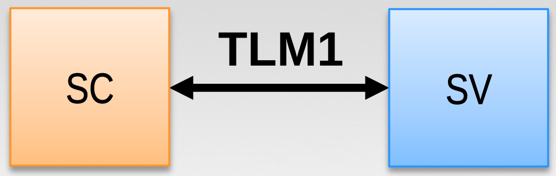
VERIFICATION ACADEMY

UVM Connect Presentation Series

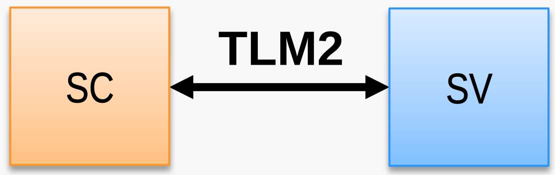
- **Part 1 – UVMC Introduction**
  - Learn what UVMC is and why you need it
  - Review the principles behind the TLM1 and TLM2 standards
  - Review basic port/export/interface connections in both SC and SV
- **Part 2 – UVMC Connections**
  - Learn how to establish connections between TLM-based components in SC and SV
- **Part 3 – UVMC Converters**
  - Learn how to write the converters that are needed to transfer transaction data across the language boundary
- **Part 4 – UVMC Command API**
  - Learn how to access and control key aspects of UVM simulation from SystemC

What is UVM Connect?

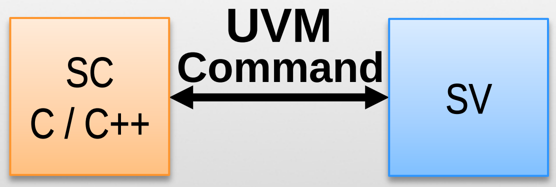
- **Connect SC & SV Models Using TLM1**



- **Connect SC & SV Models Using TLM2**

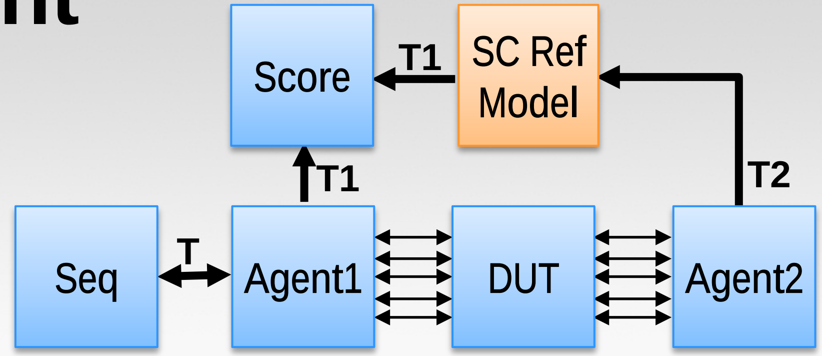


- **UVM Query and Control via Command API**

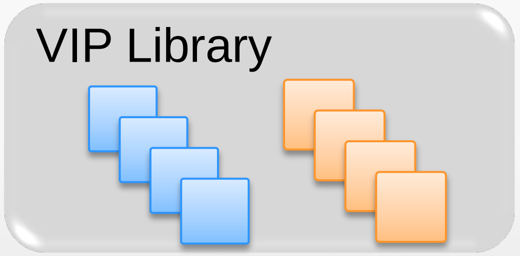


What is UVM Connect for?

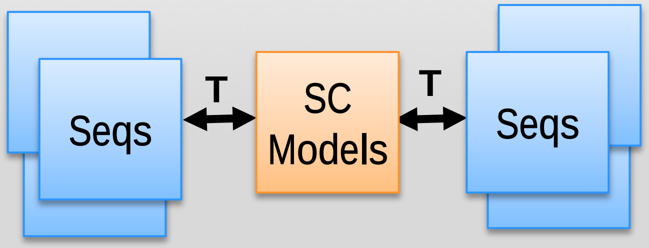
- **Abstraction Refinement**
  - Reuse SC models as reference models in SV
  - Integrate SV RTL models in SC



- **Expand VIP Inventory**
  - Integrate more off-the-shelf VIP

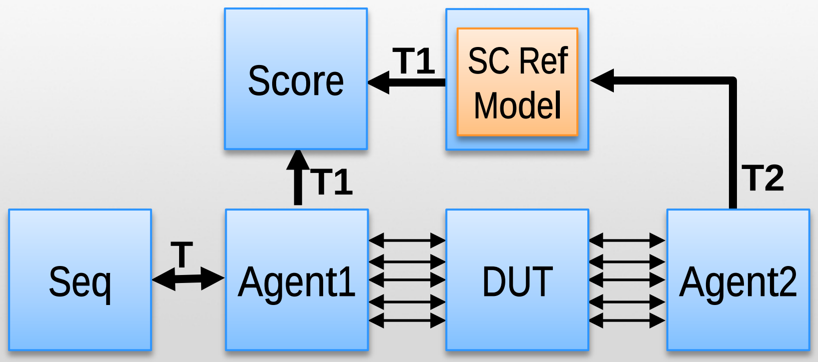


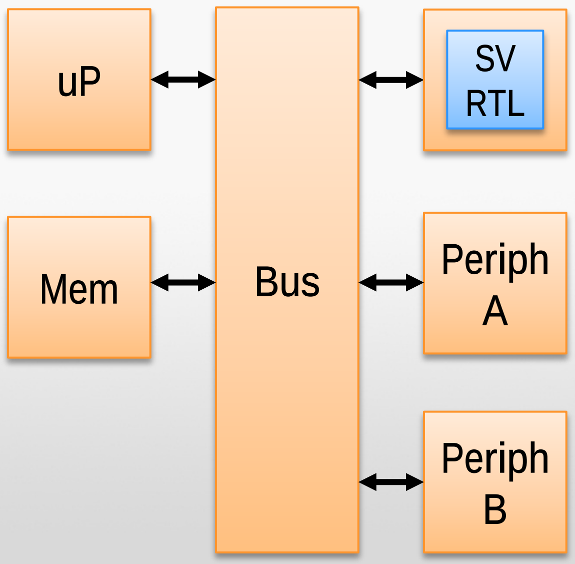
- **Leverage lang strengths**
  - Drive SC models with random stimulus from SV



What is UVM Connect for?

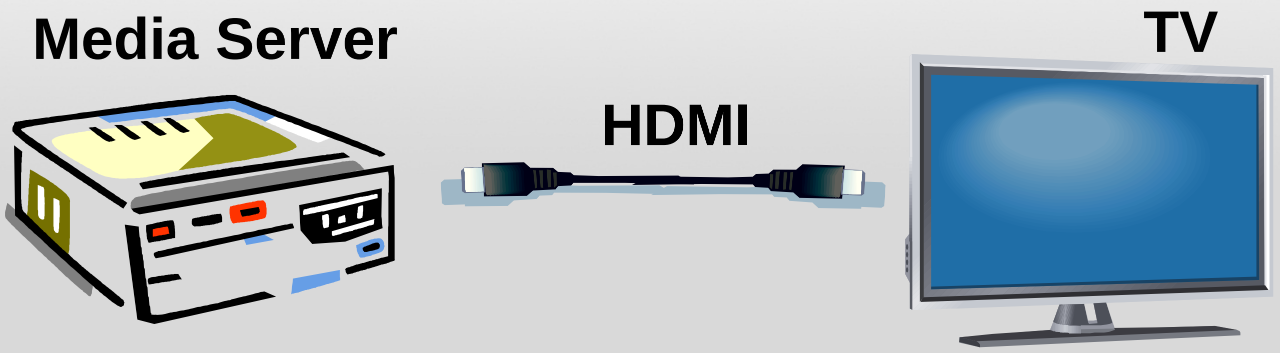
- **Encapsulation**
  - Minimize modifications when substituting components
  - Wrap foreign IP in native skin
    - integrate as ordinary native component
  - Hide cross-language connection details





Interoperability Using Standard Interfaces

- **To be reusable & easy to use, components must be**
  - Independent of their context, not expose implementation
- **To interoperate, components must agree on**
  - Information to exchange (i.e. the data type)
  - Means of exchanging that information (i.e. the interface)
- **Analogy : Media Server and TV**
  - They don’t know about each other; designed independently
  - They agree on common data (video) & interface (HDMI)
  - ***Both can be connected to many other devices***



Interoperability Using Standard Interfaces

- **Interfaces**
  - An interface is a group of methods with well-defined semantics
  - Hide implementation
    - Implementation can change without affecting code that uses it
    - Also known as “encapsulation” or “decoupling”
  - Make integration easier—connect the ports, like modules
  - Enhance reuse—can reuse components in may different contexts

- **Transaction-Level Interfaces**
  - The *interface* conveys transaction objects
  - “Initiator” components call these methods to send or request new transactions
  - “Target” components implement these methods to execute transactions or fulfill requests
  - TLM ports, exports, and sockets isolate interface callers from interface implementors

TLM Standard Interfaces

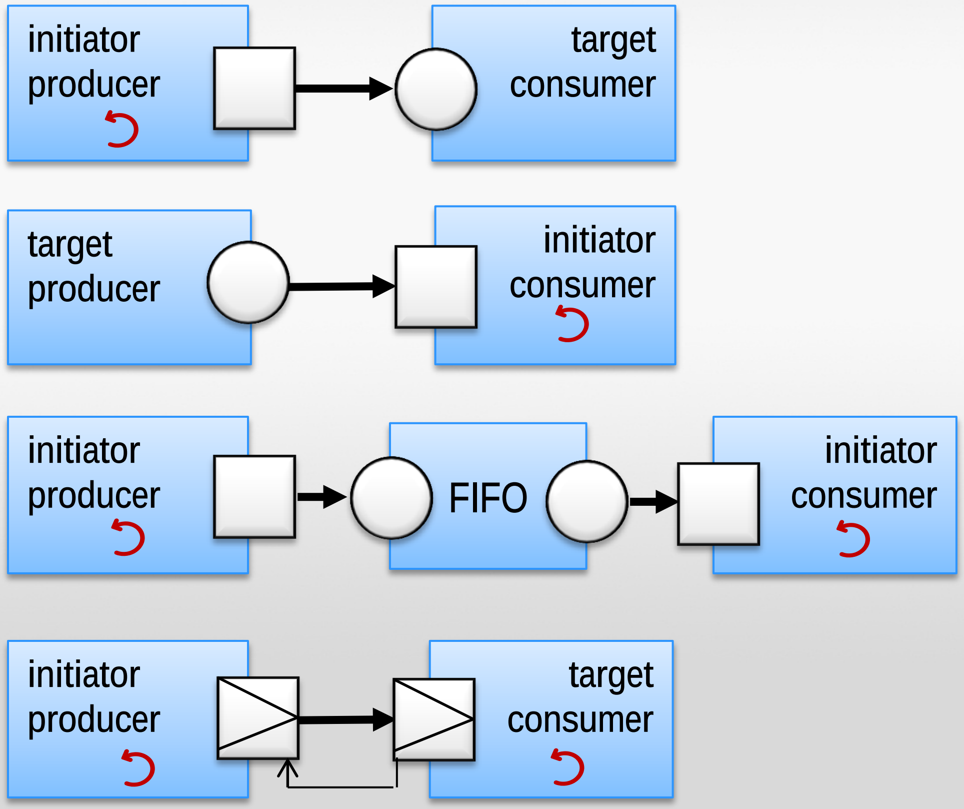
- **TLM 1.0 - 2005**
  - Defines simple intput, get, peek, analysis, fifo
  - Semantics only loosely defined
  - pass by value

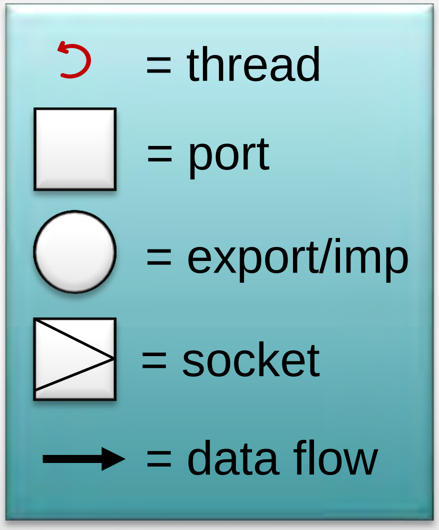
- **TLM 2.0 - 2009**
  - Much stronger standard with well-defined semantics
  - 190-page LRM, now part of IEEE 1666 (SystemC) LRM
  - optimized for high-speed, memory-mapped bus interfaces
  - b_transport, nb_transport_fw|bw, sockets
  - tlm_generic_payload (canonical trans type) and protocol
  - pass by reference

- **Implementations in SC and SV UVM**
  - TLM library in SC *is* fully compliant, complete
  - TLM library in UVM is an approximation
    - constrained by limitations in SV, “80%” complete

Ports, Exports, and Interfaces

- TLM port connections are like Verilog module port connections, except we’re connecting interfaces, not wires.
  - Initiators→initiate requests, Targets→ satisfy requests (control flow)
  - Producers→produce data, Consumers→execute data (data flow)

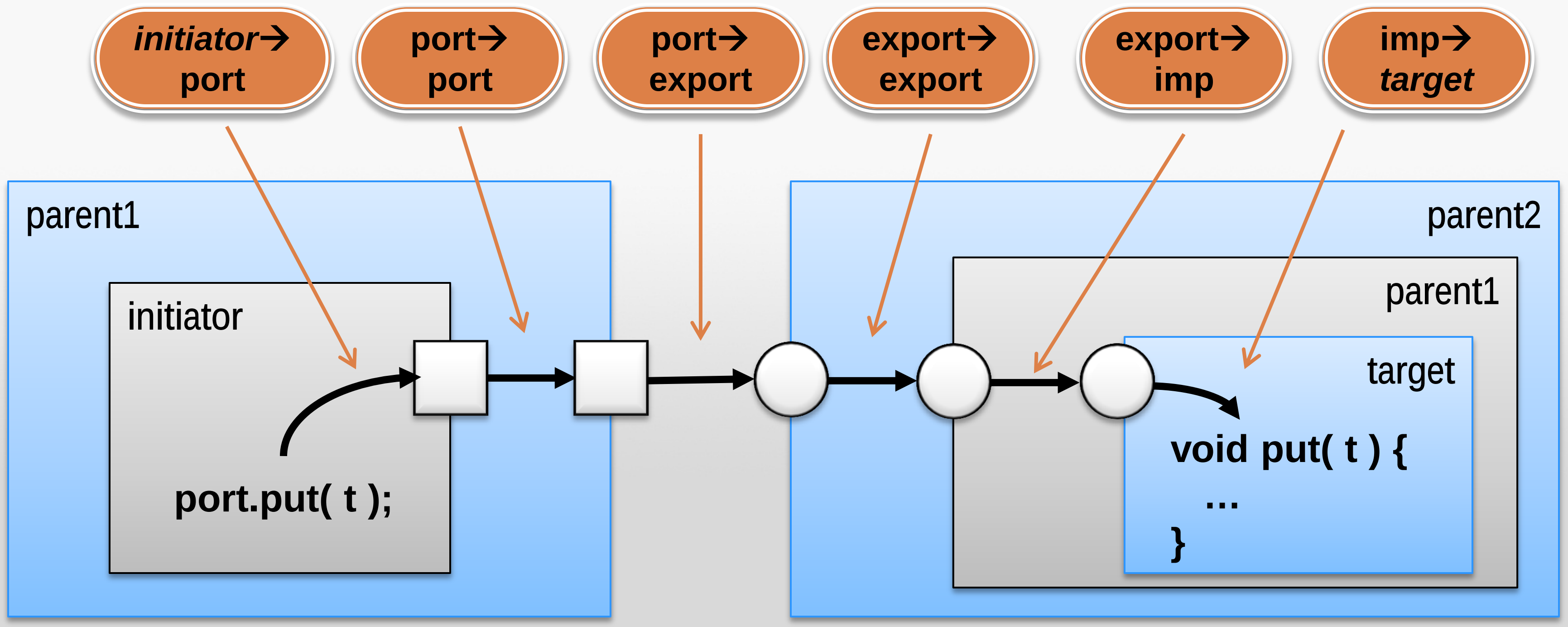




Hierarchical TLM Connections

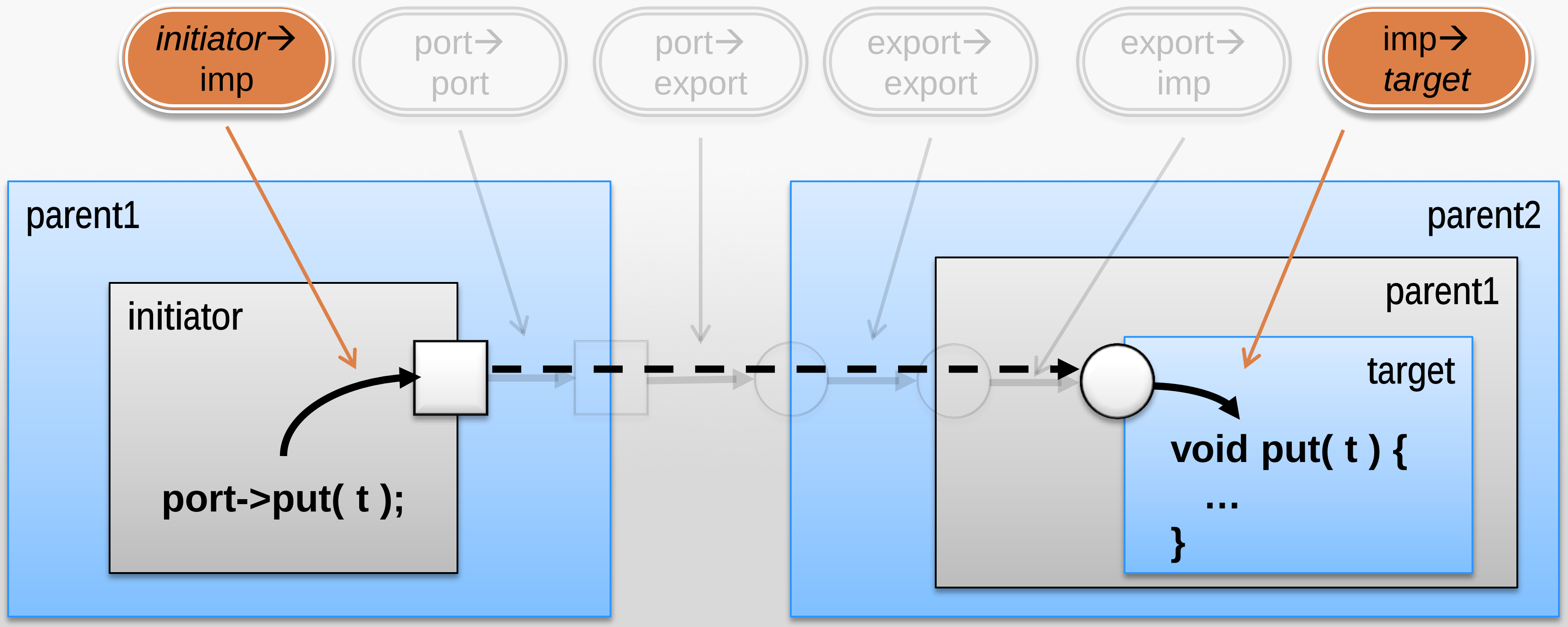
- **Ports can connect/bind to**
  - Parent ports
  - Sibling exports
  - Sibling interfaces/imps
- **Exports can connect/bind to**
  - Child exports
  - Child interfaces/imps

- **Interfaces (SC)**
  - Pure virtual classes inherited by target. No explicit binding.
- **Imps (SV)**
  - Are implicitly bound to target when allocated in target's c'tor
    - in = new("in", **this**);

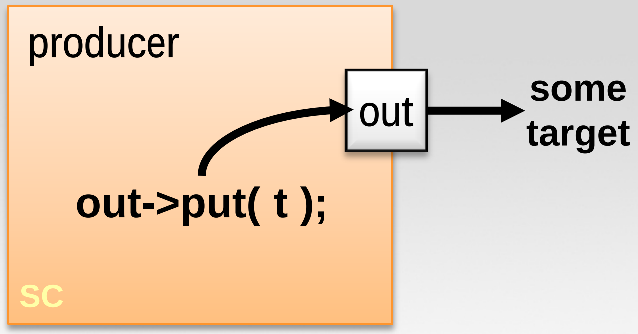


Hierarchical TLM Connections

- **Resolving port connections**
  - occurs just before end of elaboration
  - all ports requiring connections connected?
    - SC and SV do this checking independently
  - if all OK, port-export-interface/imp network collapsed such that calls incur at most 2 hops
    - SC: initiator call → port → target method
    - SV: initiator call → port → imp → target method



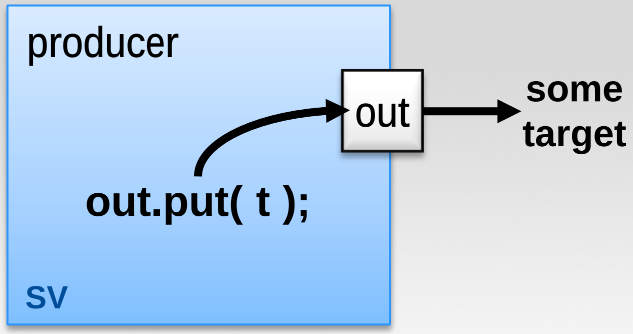
TLM Ports



- Ports are used to call interface methods implemented elsewhere
- The “starting point” for communication by initiators
- Integrator connects port *externally*
- Depicted as square in diagrams ☐

```
class producer : public sc_module
{
  sc_port<tlm_blocking_put_if<packet> > out;

  producer(sc_module_name nm) : out(“out”){
     SC_THREAD(run);
  }
  void run() {
     packet t;
     ...initialize/randomize packet...
     out->put( t );
  }
};
```

TLM Ports



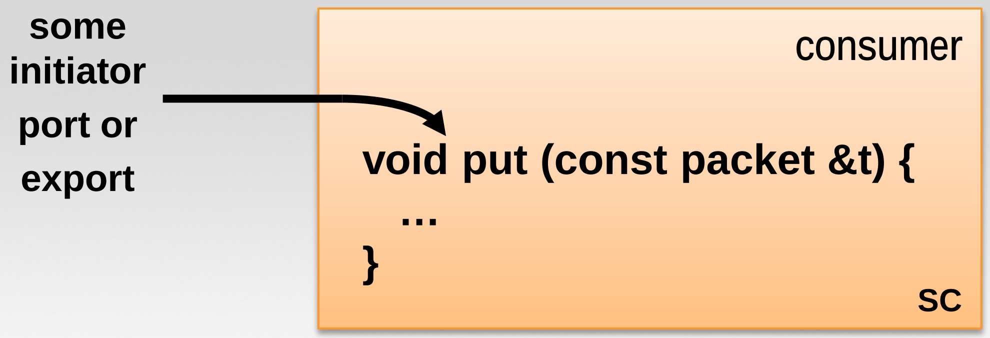
- Ports are used to call interface methods implemented elsewhere
- The “starting point” for communication by initiators
- Integrator connects port *externally*
- Depicted as square in diagrams

```
class producer extends uvm_component {
  tlm_blocking_put_port #(packet) out;

  function new (string name, uvm_component parent=null);
    super.new(name,parent);
    out = new(“out",  this);
  endfunction

  virtual task run_phase (uvm_phase phase);
    packet t;
    ...initialize/randomize packet...
     out.put(t);
   endtask
};
```

TLM Interfaces and Imps

- Are the “end point” in a network of port-export-interface/imp connections

- In SC, the target component inherits the *interface* & implements its methods. Target can inherit (and implement) multiple interfaces.



```
void put (const packet &t) {
  …
}
```

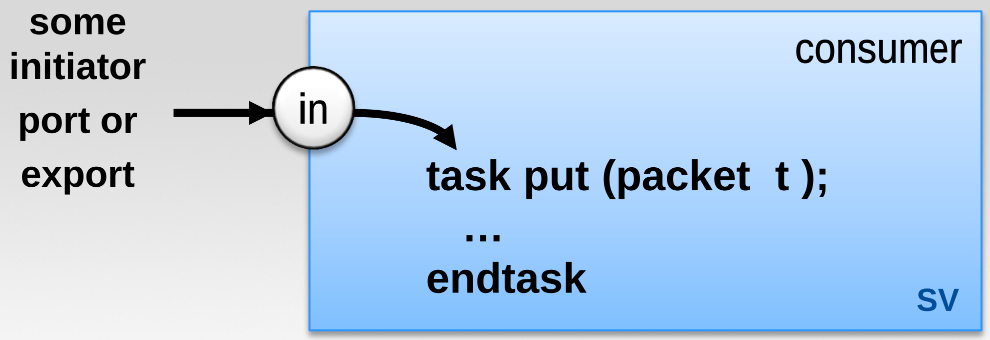
```
class consumer : public sc_module,
                 tlm_blocking_put_if<packet>
{
  public:
  consumer(sc_module_name nm) {
  }

  virtual void put(const packet &t) {
    cout << “Got packet: “ << t << endl;
    wait(10,SC_NS);
  }
};
```

TLM Interfaces and Imps

- Are the "end point" in a network of port-export-interface/imp connections

- In SC, the target component inherits the *interface* & implements its methods. Target can inherit (and implement) multiple interfaces.

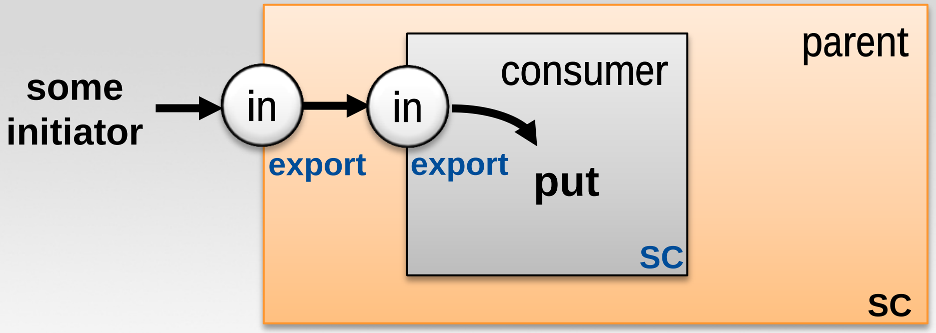
- In SV, no multiple inheritance. Uses *imps* as workround. *Imps* delegate external calls to the component that implements them. Depicted in diagrams as circle



```
class consumer extends uvm_component;
  uvm_blocking_put_imp #(packet,consumer) in;
  `uvm_component_utils(consumer)
   function new(string name,
                uvm_component parent=null);
      super.new(name,parent);
      in = new("in",  this);
   endfunction

   virtual task put (packet t);
     `uvm_info("CONSUMER/PKT/RECV",
              t.sprint(),UVM_MEDIUM)
     #10ns;
   endtask
endclass
```

TLM Exports

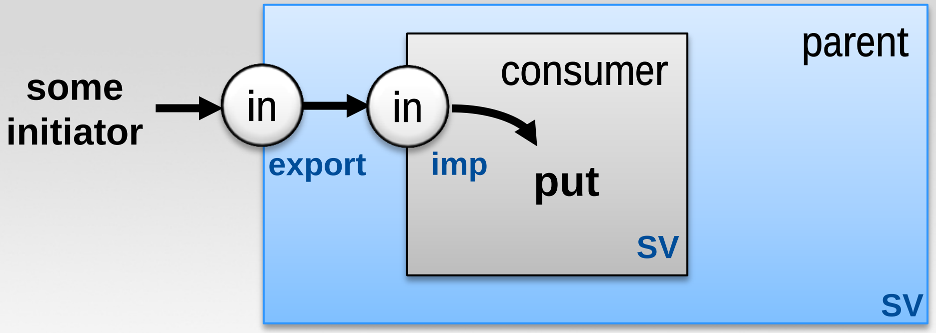
- **Promotes an interface implementation from a child to parent**
- **Internally bound to child export or imp/interface in constructor**
- **Externally connected to port or parent export, but not required. (If no connection, no activity)**
- **Depicted as circle in diagrams**



```
class consumer : public sc_module,
                 tlm_blocking_put_if<packet>
{
  public:
  sc_export<tlm_blocking_put_if<packet> > in;

  consumer(sc_module_name nm) : in("in") {
    in(*this); // promote own intf impl
  }

 virtual void put(const packet &t) {
    cout << “Got packet: “ << t << endl;
    wait(10,SC_NS);
  }
};
```

TLM Exports

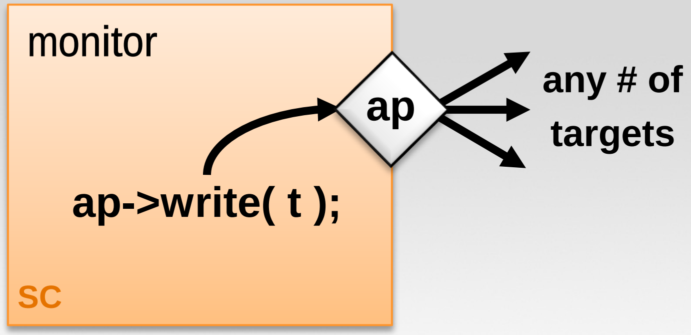
- Promotes an interface implementation from a child to parent
- Internally bound to child export or imp/interface in constructor
- Externally connected to port or parent export, but not required. (If no connection, no activity)
- Depicted as circle in diagrams



```
class parent extends uvm_component;
  uvm_blocking_put_export #(packet) in;
  consumer cons;
  `uvm_component_utils(consumer)
  function new(string name,
                 uvm_component parent=null);
     super.new(name,parent);
     in = new("in",  this);
  endfunction
  function void build_phase(uvm_phase phase);
    cons=consumer::type_id::create(“const”,this);
  endfunction
  function void connect_phase(uvm_phase phase);
    in.connect(consumer.in);
  endfunction
endclass
```

TLM Analysis Ports

- **Ports that broadcast to all connected targets (publisher/subscriber)**
- **Read-only. Targets must not modify. For debug, scoreboards, etc.**
- **Usually does not *require* connection** (SC_ZERO_OR_MORE_BOUND)
- **Depicted as diamond in diagrams**

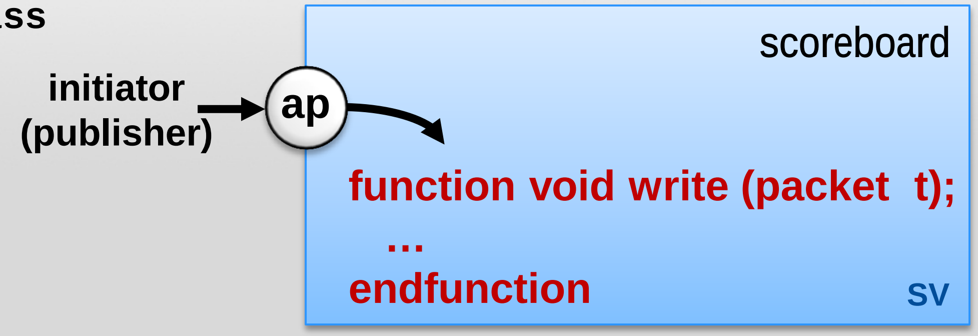


```
class monitor: public sc_module
{
  sc_port<tlm_analysis_if<packet>,
         0,SC_ZERO_OR_MORE_BOUND> > ap;
  monitor(sc_module_name nm) : ap(“ap”){
     SC_THREAD(run);
  }
  void run() {
     packet t;
     ...capture packet off bus...
     ap->write( t );
  }
};
```

TLM Analysis Imps

- Receives streams of transactions from connected publisher (e.g. monitor)
- Read-only. For debug, scoreboards, etc.
- Like all exports/imps, does not require a connection
- Depicted as circle in diagrams

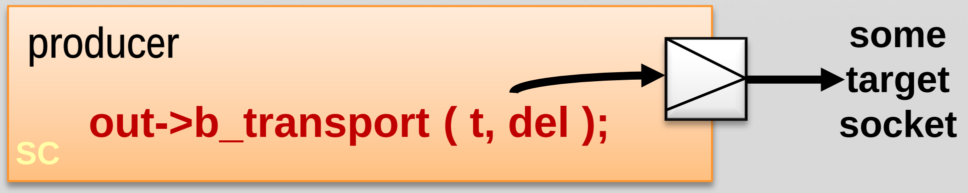
```
class scoreboard extends uvm_component;
   uvm_analysis_imp #(packet,scoreboard) actual_in;
   `uvm_component_utils(scoreboard)
   function new(string name,
                uvm_component parent=null);
      super.new(name,parent);
      actual_in = new("actual_in", this);
      ...
   endfunction
   virtual function void write(packet t);
     packet exp;
     `uvm_info("SB/PKT/RECV",t.sprint(),UVM_MEDIUM)
     if (!expect_fifo.try_get(exp)) ...error
     if (!t.compare(exp)) ...error
   endfunction
endclass
```



TLM Initiator Socket

- **Can do blocking or non-blocking transport (Usually one or the other)**
- **Default type parameters → tlm_generic_payload and base protocol**
- **Initiator *must* implement all of bw interface**
  - unless simple initiator socket used (utility layer)
- **If driving SV target, mem and debug methods can be stubbed out; they won't be called**
- **Depicted as square with outward facing arrow**

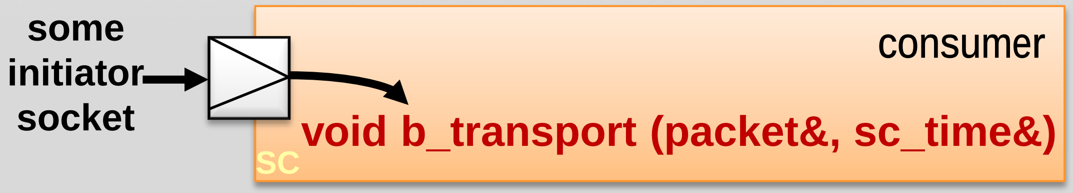
```
struct producer: public sc_module,
                 public tlm_bw_transport_if< >
{
  tlm::tlm_initiator_socket< > out; // tlm_gp
  producer (sc_module_name nm) : out(“out") {
    out(*this); // bind bw intf to self
    SC_THREAD(fw_proc);
  }
  // FORWARD PATH
  void fw_proc() {
    // prepare tlm gp trans...
    out->b_transport(t,del); *or*
    out->nb_transport_fw(t,ph,del);
  }
  // BACKWARD PATH
  virtual tlm_sync_enum nb_transport_bw(...) {
     ...coordinate with fw path, per protocol
  }
  virtual void invalidate_direct_mem_ptr(...) {
    // Dummy implementation
  }
};
```



TLM Target Socket

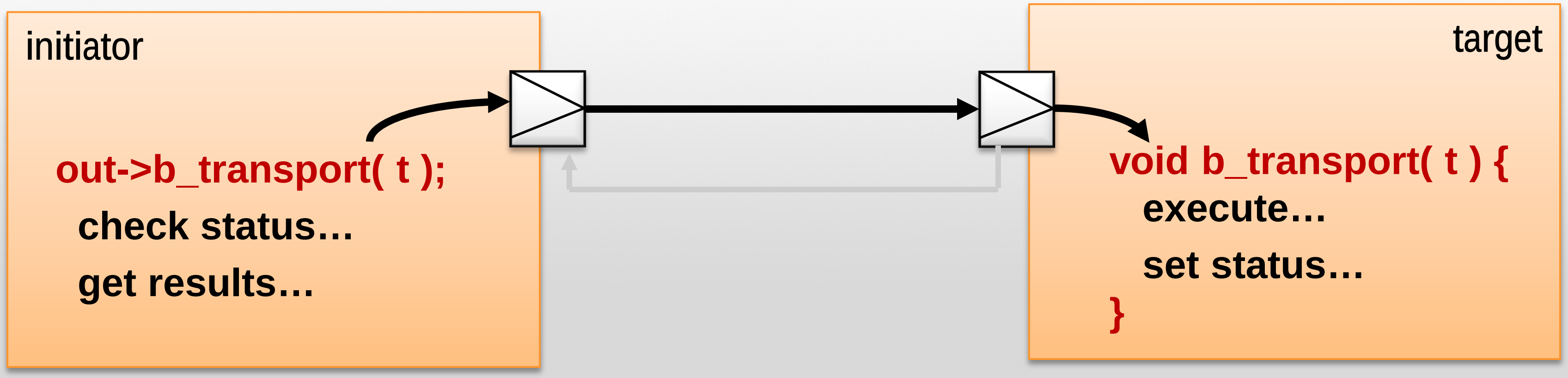
- Can impl blocking or non-blocking transport (Usually one or the other)
- Default type parameters → tlm_generic_payload and base protocol
- Target *must* implement all of fw interface
  - unless simple target socket used (from utility layer)
- If driving SV target, mem and debug methods can be stubbed out.
- Depicted as square with inward facing arrow

```
struct consumer: public sc_module,
                 public tlm_fw_transport_if< > {
  tlm::tlm_target_socket< > in;
  consumer(sc_module_name nm) : in(“in") {
    in.bind(*this);
    SC_THREAD(bw_proc);
  }
  // FORWARD PATH
  void b_transport( packet& trans,sc_time& t ) {
    // fully execute request, modify args, return }
  tlm_sync_enum nb_transport_fw(...) {
    // per protocol, update args as allowed,return}
  bool get_direct_mem_ptr(…) { return FALSE; }
  uint transport_dbg(…) { return 0; }

  // BACKWARD PATH
  void bw_proc() {
    ...coordinate with fw transport per protocol
    in->nb_transport_bw(trans,ph,delay);
  }
};
```



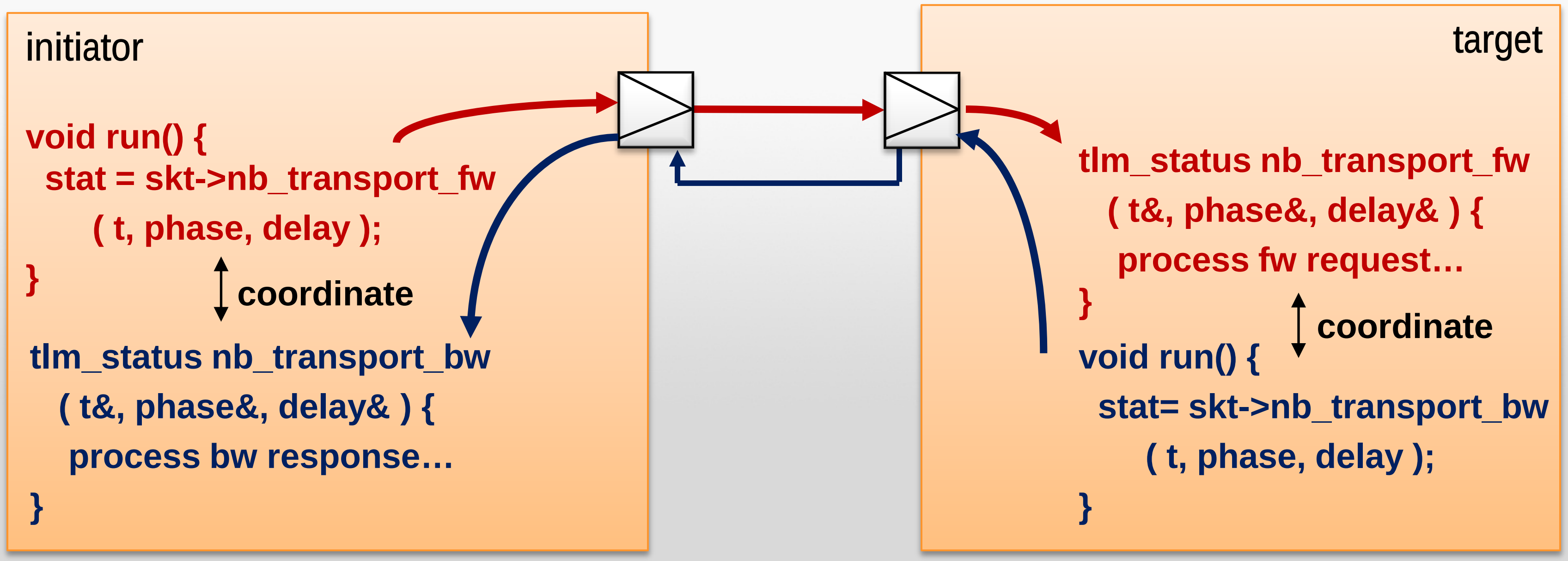
TLM Sockets – Blocking Transport

- **Blocking Transport**
  - Initiator *indirectly* calls **b_transport** in Target
  - Initiator must not modify transaction; transaction contents invalid until **b_transport** returns
  - When **b_transport** returns, transaction is complete with status/results
  - Transaction can be reused in next **b_transport** call



TLM Sockets – Non-blocking Transport

- **Non-blocking Transport using Base Protocol**
  - Initiator starts request by calling nb_transport_fw in Target
    - Target returns with updated arguments
  - Target can call nb_transport_bw in Initiator at phase transitions
    - To provide Initiator updates; Initiator may respond via fw interface
  - Transaction contents, phase, & delay can change
    - Only certain fields in certain phases, according to base protocol rules
  - Transport calls continue back and forth until either returns transaction complete status
    - Same transaction handle used throughout its execution, for efficiency
    - concurrent transactions possible—using different transaction object



```
void run() {
  stat = skt->nb_transport_fw
      ( t, phase, delay );
}

tlm_status nb_transport_bw
   ( t&, phase&, delay& ) {
    process bw response…
}
```

```
tlm_status nb_transport_fw
  ( t&, phase&, delay& ) {
    process fw request…
}
void run() {
  stat= skt->nb_transport_bw
      ( t, phase, delay );
}
```

TLM Generic Payload

- **TLM 2.0 defines**
  - a base transaction type: tlm_generic_payload (TLM GP)
  - a base protocol with initiator/target sockets

- **When used together, interoperability is maximized**

- **The TLM GP definitions and converters are built-in**
  - models that use the TLM GP are the easiest to integrate
  - *connect and go!*

Field	Description
command	READ, WRITE, or IGNORE
address	Base address
data	Data buffer, array of bytes.
data_length	Number of valid bytes in data buffer
response_ status	OK, INCOMPLETE, GENERIC_ERROR, ADDRESS_ERROR, BURST_ERROR,etc.
byte_enable	Byte-enable data
byte_enable_ length	Number of valid byte-enables

Summary

- **UVMC provides TLM1 and TLM2 connectivity SC←→SV**
  - See Part 2: UVMC Connections
  - See Part 3: UVMC Converters
- **UVMC provides a UVM Command API**
  - For accessing and controlling UVM simulation from SC
  - See Part 4: UVMC Command API
- **TLM helps isolate component implementations from the outside world**
  - Integration easier—connect the ports, like modules
  - Reuse enhanced—can reuse components in may different contexts
- **TLM interoperability requires that components agree on**
  - data to exchange
  - method of exchange (interface)
  - direction of exchange (initiator or target)

UVM Connect Presentation Series

- **Part 1 – UVMC Introduction**
  - Learn what UVMC is and why you need it
  - Review the principles behind the TLM1 and TLM2 standards
  - Review basic port/export/interface connections in both SC and SV
- **Part 2 – UVMC Connections**
  - Learn how to establish connections between TLM-based components in SC and SV
- **Part 3 – UVMC Converters**
  - Learn how to write the converters that are needed to transfer transaction data across the language boundary
- **Part 4 – UVMC Command API**
  - Learn how to access and control key aspects of UVM simulation from SystemC

UVM Connect

Part 1 – Introduction

Adam Erickson
Verification Technologist

academy@mentor.com
www.verificationacademy.com

VERIFICATION ACADEMY