Mentor Graphics®

UVM Connect

Part 3 – Converters

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

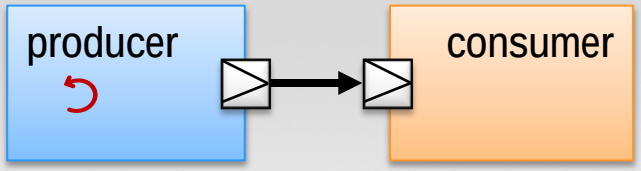
Agenda

- **Behind the scenes**
  - A look at how the converters are used
- **Default Converters**
  - For SC and SV; Built in support for tlm_generic_payload
- **SV Converter Options**
  - Implement inside or outside transaction class
- **SC Converter Options**
  - Implement inside or outside transaction class
  - Adapting while converting

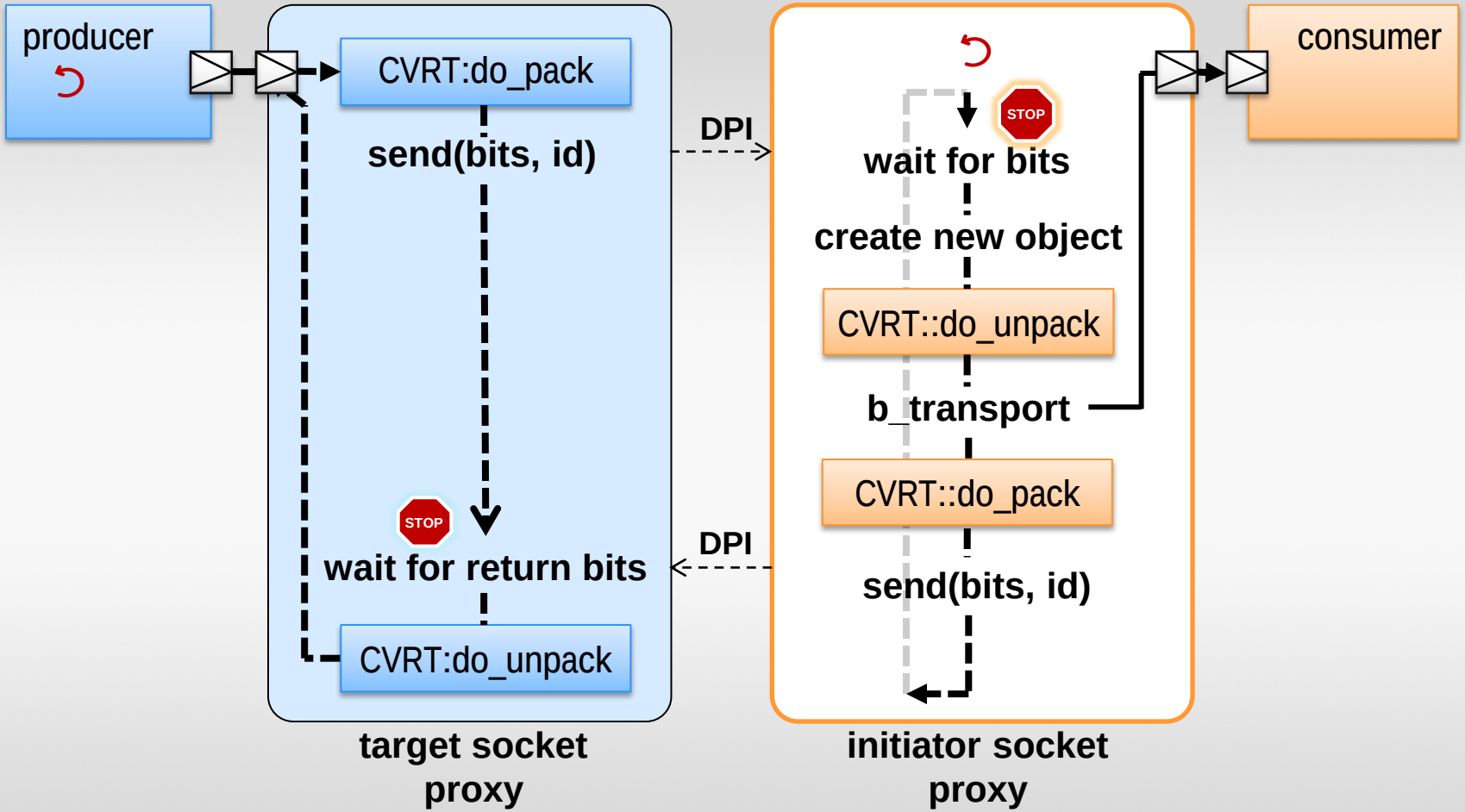
Agenda

- **Behind the scenes**
  - A look at how the converters are used
- **Default Converters**
  - For SC and SV; Built in support for tlm_generic_payload
- **SV Converter Options**
  - Implement inside or outside transaction class
- **SC Converter Options**
  - Implement inside or outside transaction class
  - Adapting while converting

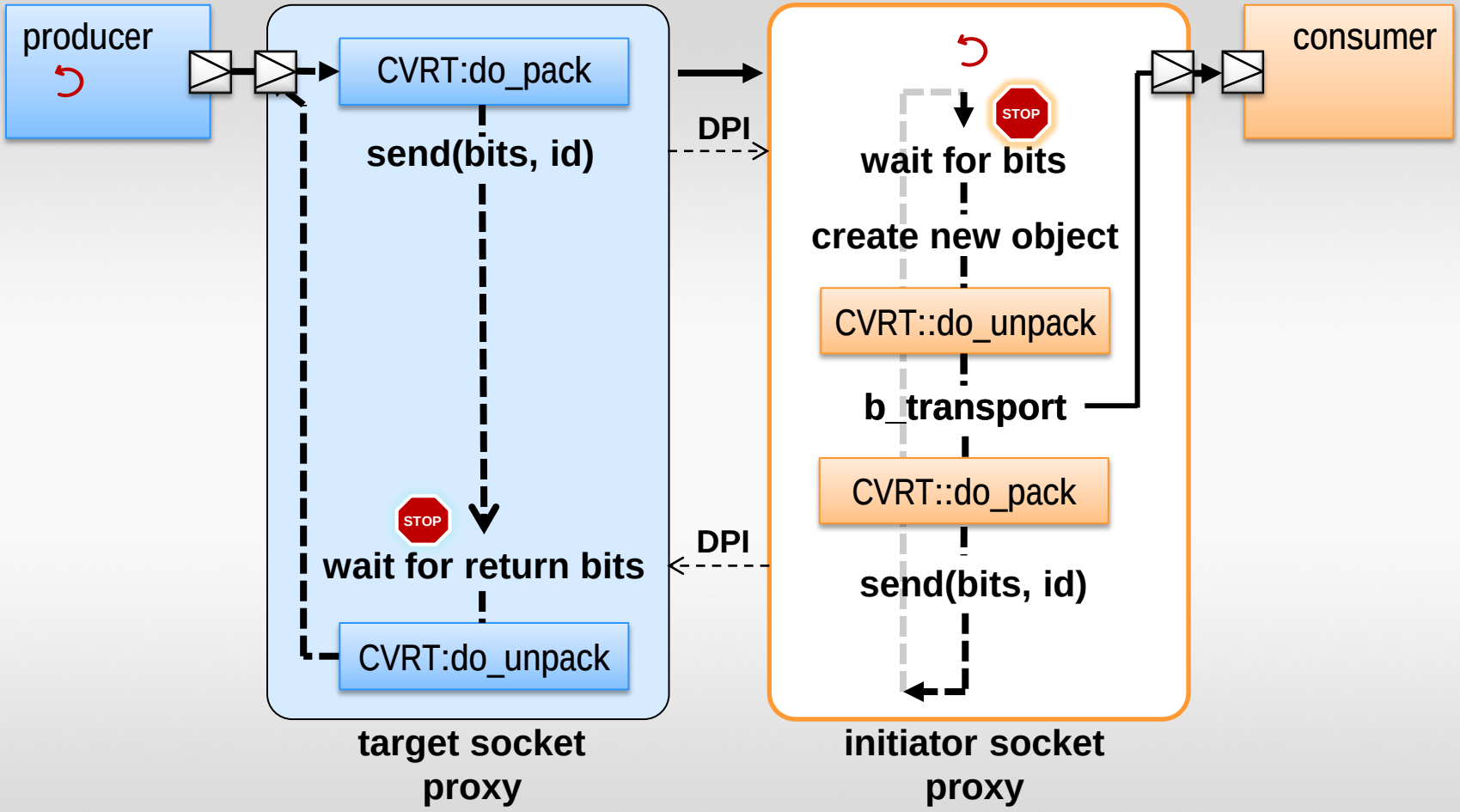
Behind the Scenes



Behind the Scenes



Behind the Scenes

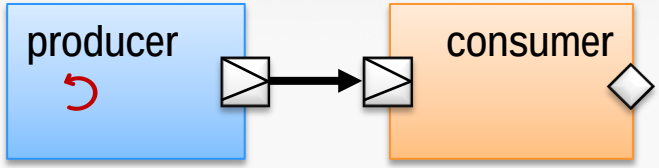


Agenda

- Behind the scenes
  - A look at how the converters are used
- **Default Converters & Type Support**
  - For SC and SV
  - Built in support for tlm_generic_payload
- SV Converter Options
  - Implement inside or outside transaction class
- SC Converter Options
  - Implement inside or outside transaction class
  - Adapting while converting

Built in Support for Generic Payload

- **Full TLM GP support is built into the libraries**
  - Components using TLM GP with base protocol are easist to integrate, most interoperable
  - In most cases TLM GP is all you should ever need and use




```
import uvm_pkg::*;
import uvmc_pkg::*;
`include "producer.sv"

module sv_main;
 producer prod = new("prod");
 initial begin
  uvmc_tlm #()::connect(prod.out,
                        "foo");
  run_test();
 end
endmodule
```

Uses default uvm_tlm_generic_payload

```
#include "uvmc.h"
using namespace uvmc;
#include "consumer.h"

int sc_main(int argc, char* argv[])
{
  consumer cons("cons");
  uvmc_connect(cons.in,"foo");
  sc_start();
  return 0;
}
```

Type Support

Category	SV	SC
Signed integrals	longint int shortint byte bit	long long* int short char bool
Unsigned integrals	longint unsigned int unsigned shortint unsigned byte unsigned bit unsigned	unsigned long long* unsigned int unsigned short unsigned char bool
Misc	shortreal real string time enum	float double string sc_time enum
Arrays	T arr[N] T q[$] T da[] T aa[KEY]	T arr[N] vector<T> list<T> map<KEY,T>
Bit vectors	bit logic bit [L:R] logic [L:R]	sc_bit sc_logic sc_bv<N> sc_lv<N>
SC integers	bit [N-1:0] bit [N-1:0] bit [N-1:0] bit [N-1:0]	sc_int<N> sc_uint<N> sc_bigint<N> sc_biguint<N>

The UVM uvm_packer and UVMC uvmc_packer support these types directly.

Other types can be accommodated in your converters

Sub-objects supported. Just call their converter methods directly

On 32-bit machines, SC long long is 32 bits. SV longints are always 64 bits regardless of architecture

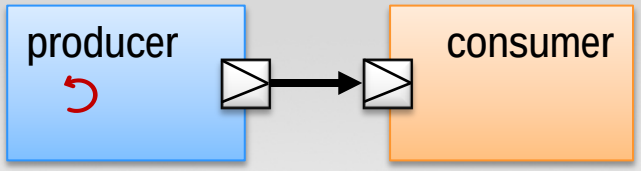
Transaction Used in non-GP Examples

3 Data Members:

- **Command**
  - 32-bit enum: { READ, WRITE, NOOP }
- **Address**
  - 32-bit Integer
- **Data**
  - Variable array of bytes

Default Converters

Both delegate to T.pack/unpack



The SC default is rarely appropriate.

```
class uvmc_default_converter
  #(type T=int) extends
      uvmc_converter #(T);

 static function
  void do_pack (T t,
                uvm_packer packer);
    t.pack(packer);
 endfunction

 static function
   void do_unpack (T t,
                   uvm_packer packer);
    t.unpack(packer);
endfunction

endclass
```

```
template <typename T>
class uvmc_converter {
  public:

  static void
     do_pack(const T &t,
             uvmc_packer &packer) {
     t.do_pack(packer);
  }

  static void
    do_unpack(T &t,
              uvmc_packer &packer) {
     t.do_unpack(packer);
  }

};
```

Agenda

- **Behind the scenes**
  - A look at how the converters are used
- **Default Converters**
  - For SC and SV; Built in support for tlm_generic_payload
- **SV Converter Options**
  - Implement inside or outside transaction class

3 ways

- **SC Converter Options**
  - Implement inside or outside transaction class
  - Adapting while converting

SV Conversion – do_pack/unpack

```
class packet extends uvm_sequence_item;
  typedef enum { WRITE, READ, NOOP } cmd_t;
  rand cmd_t cmd;
  rand int   addr;
  rand byte  data[$];

  function new(string name="");
    super.new(name);
  endfunction

  `uvm_object_utils(packet)
  virtual function void do_pack (uvm_packer packer);
    `uvm_pack_enum(cmd)
    `uvm_pack_int(addr)
    `uvm_pack_queue(data)
  endfunction
  virtual function void do_unpack (uvm_packer packer);
    `uvm_unpack_enum(cmd,cmd_t)
    `uvm_unpack_int(addr)
    `uvm_unpack_queue(data)
  endfunction
endclass
```

extends uvm_sequence_item

conversion functions are implemented as methods of the transaction class

define *do_pack* using `uvm_pack_* macros

define *do_unpack* using `uvm_unpack_* macros

SV Conversion – Field Macros

```
class packet extends uvm_sequence_item;
  typedef enum { WRITE, READ, NOOP } cmd_t;
  rand cmd_t cmd;
  rand int   addr;
  rand byte  data[$];

  function new(string name="");
    super.new(name);
  endfunction

  `uvm_object_utils_begin(packet)
    `uvm_field_enum(cmd_t,cmd,UVM_ALL_ON)
    `uvm_field_int(addr,UVM_ALL_ON)
    `uvm_field_queue_int(data,UVM_ALL_ON)
  `uvm_object_utils_end

   ...



endclass
```

extends uvm_sequence_item

conversion functions are implemented via *`uvm_field* macros

`uvm_field macros degrade overall performance

`uvm_field macros hinder debug

SV Conversion – External Converter

```
class packet;
  typedef enum { WRITE, READ, NOOP } cmd_t;
  rand cmd_t cmd;
  rand int   addr;
  rand byte  data[$];
endclass

class convert_packet extends uvmc_converter #(packet);

  static function void do_pack(packet t, uvm_packer packer);
    `uvm_pack_enum(t.cmd)
    `uvm_pack_int(t.addr)
    `uvm_pack_queue(t.data)
  endfunction

  static function void do_unpack(packet t, uvm_packer packer);
    `uvm_unpack_enum(t.cmd,packet::cmd_t)
    `uvm_unpack_int(t.addr)
    `uvm_unpack_queue(t.data)
  endfunction

endclass
```

no base class (or doesn't extend uvm_object)

custom converter must extend *uvmc_converter #(T)*

define *do_pack*

using same `uvm_pack_* macros as for in-transaction do_pack

define *do_unpack*

using same `uvm_unpack_* macros as for in-transaction do_unpack

Agenda

- **Behind the scenes**
  - A look at how the converters are used
- **Default Converters**
  - For SC and SV; Built in support for tlm_generic_payload
- **SV Converter Options**
  - Implement inside or outside transaction class
- **SC Converter Options**
  - Implement inside or outside transaction class
  - Adapting while converting

4 ways

SC Converter Specialization

```
class packet {
  enum cmd_t { WRITE=0, READ, NOOP };
  cmd_t cmd;
  unsigned int addr;
  vector<unsigned char> data;
};

template <>
struct uvmc_converter<packet_base> {

  static void do_pack (const packet_base &t,
                       uvmc_packer &packer) {
    packer << t.cmd << t.addr << t.data;
  }

  static void do_unpack (packet_base &t,
                         uvmc_packer &packer) {
    packer >> t.cmd >> t.addr >> t.data;
  }
};
```

no base class.
no dependencies.

Define template specialization of *uvmc_converter<T>*

Define *do_pack* .
Stream from data members into packer using operator <<

Define *do_unpack*.
Stream into data members from packer using operator >>

Conversion is simple: just “stream” all your fields to/from the packer.

SC Converter Specialization - Macros

```
class packet {
  enum cmd_t { WRITE=0, READ, NOOP };
  cmd_t cmd;
  unsigned int addr;
  vector<unsigned char> data;
};

template <> struct uvmc_converter<packet> {
  static void do_pack(const packet &t, uvmc_packer &packer) {
    packer << t.cmd << t.addr << t.data;
  }
  static void do_unpack(packet &t, uvmc_packer &packer) {
    packer >> t.cmd >> t.addr >> t.data;
  }
};
UVMC_UTILS_3(packet,cmd,addr,data)
```

define this exact template specialization of *uvmc_converter<T>* by invoking one *UVMC_UTILS_x* macro (*x = 1 to 20*)

These are “good” macros—they expand into code that you would write

SC Converter – In Class

```
class packet {
  enum cmd_t { WRITE=0, READ, NOOP };
  cmd_t cmd;
  unsigned int addr;
  vector<unsigned char> data;

  virtual void do_pack(uvmc_packer &packer) const {
    packer << cmd << addr << data;
  }

  virtual void do_unpack(uvmc_packer &packer) {
    packer >> cmd >> addr >> data;
  }

};
```

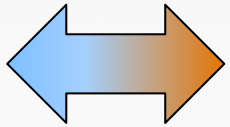
Although easy with UVMC, this is uncommon, as C++ affords better decoupling through separate converter classes.

Converting REALLY Different Types

- **Question: What if your SC and SV transactions are already fixed (and very different)?**

SV

```
class packet extends uvm_sequence;
  typedef enum { WRITE, READ, NOOP } cmd_t;
  rand cmd_t cmd;
  rand int   addr;
  rand byte  data[$];
endclass
```



SC

```
class packet {
  short addr_hi;
  short addr_lo;
  unsigned int payload[4];
  char  len;
  bool  write;
};
```

- **Answer: Leverage the SC converter to convert *AND* adapt**
  - Define SV converter “normally”
    - Options are limited
  - Define custom SC converter
    - that derives from uvmc_converter<T>

Converting REALLY Different Types

- **Step 1: Do SV conversion as usual**

```
class packet extends uvm_sequence_item;
   typedef enum { WRITE, READ, NOOP } cmd_t;
   rand cmd_t cmd;
   rand int   addr;
   rand byte  data[$];
   `uvm_object_utils(packet)
   function new(string name="");
     super.new(name);
   endfunction
   virtual function void do_pack(uvm_packer pack
     `uvm_pack_enum(cmd)
     `uvm_pack_int(addr)
     `uvm_pack_queue(data)
   endfunction
   virtual function void do_unpack(uvm_packer pa
     `uvm_unpack_enum(cmd,cmd_t)
     `uvm_unpack_int(addr)
     `uvm_unpack_queue(data)
   endfunction
 endclass
```

Converting REALLY Different Types

- **Step 2: SC converter converts *and* adapts**

SV cmd enum → SC bool

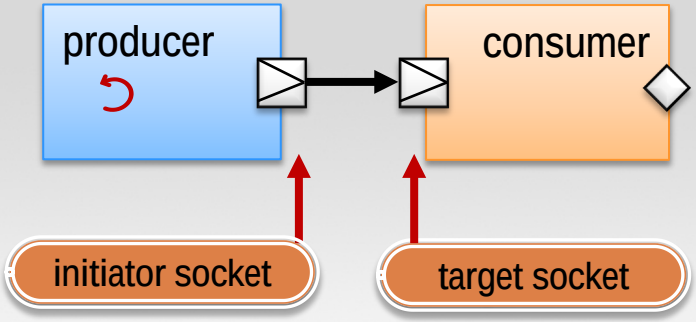
SV byte queue → SC int [4]

Adjust different streaming order

```
struct custom_packet_cvrt : public uvmc_converter<packet> {
 static void do_pack(const packet &t, uvmc_packer &packer) {
   int cmd_tmp;
   cmd_tmp = write ? 0 : 1;
   packer << cmd_tmp << t.addr_lo << t.addr_hi
          << (int)(t.len) << t.payload << (int)0;
 }
 static void do_unpack(packet &t, uvmc_packer &packer) {
   int cmd_tmp;
   vector<unsigned char> data_tmp;
   packer >> cmd_tmp >> t.addr_lo >> t.addr_hi >> data_tmp;
   t.len = data_tmp.size();
   t.write = (cmd_tmp == 0) ? 1 : 0;
   for (int i=0;i<4;i++) {
     t.payload[i]=0;
     for (int j=0;j<4;j++) {
       if ((i*4+j)<t.len) {
         int_tmp = data_tmp[i*4+j] << (8*j);
         t.payload[i] = t.payload[i] | int_tmp;
       }
       else break;
     }
   }
};
```

Converting REALLY Different Types

- **Step 3: Specify custom converter when calling connect on SC side**




```
import uvm_pkg::*;
import uvmc_pkg::*;
`include "producer.sv"

module sv_main;
 producer prod = new("prod");
 initial begin
  uvmc_tlm #()::
      connect(prod.out,"foo");
  run_test();
 end
endmodule
```

```
#include "uvmc.h"
using namespace uvmc;
#include "consumer.h"

int sc_main(int argc, char* argv[])
{
  consumer cons("consumer");
  uvmc_connect<custom_packet_cvrt>
      (cons->in,"foo");
  sc_start();
  return 0;
}
```

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 estblish 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

Mentor Graphics®

UVM Connect

Part 3 – Converters

Adam Erickson
Verification Technologist

academy@mentor.com
www.verificationacademy.com

VERIFICATION ACADEMY