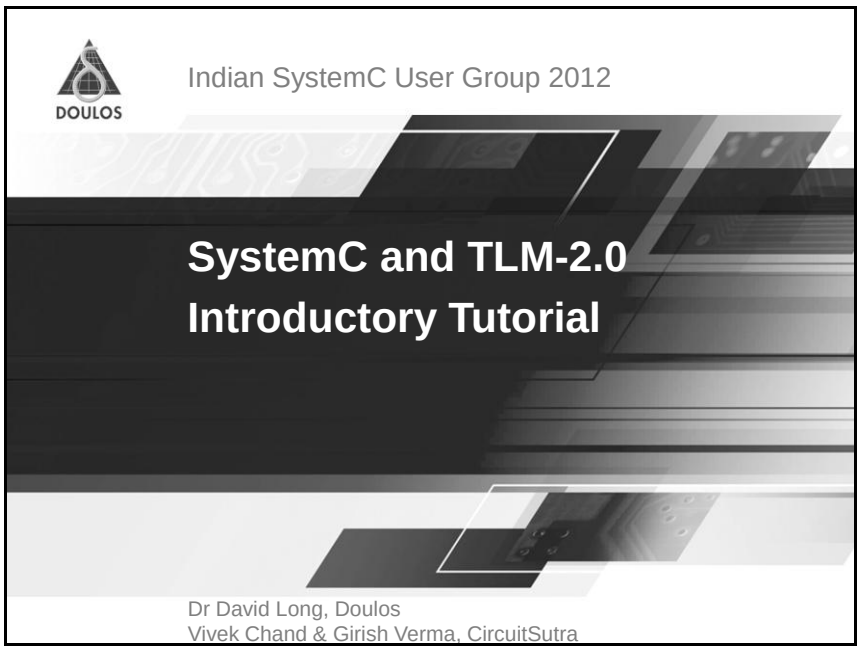


Notes

Introduction to SystemC

- Overview and background
- Central concepts
- The SystemC World
- Use cases and benefits

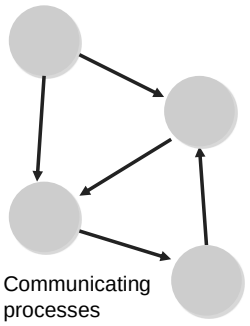
DOULOS

2

What is SystemC?

DOULOS

- System-level modeling language
  - Network of communicating processes (c.f. HDL)
  - Supports heterogeneous models-of-computation
  - Models hardware and software
- C++ class library
  - Open source proof-of-concept simulator
  - Owned by OSCI

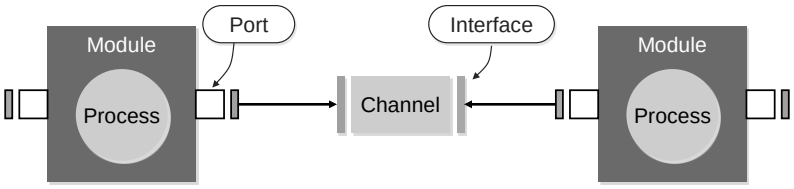


Communicating processes

3

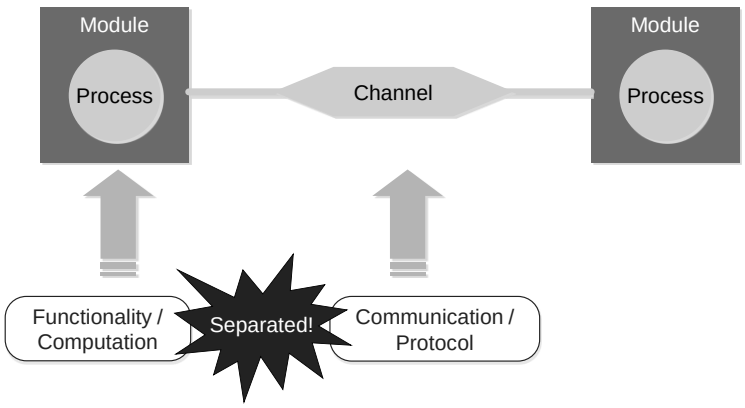
Features of SystemC

- Modules (structure)
- Ports (structure)
- Processes (computation, concurrency)
- Channels (communication)
- Interfaces (communication refinement)
- Events (time, scheduling, synchronisation)
- Data types (hardware, fixed point)



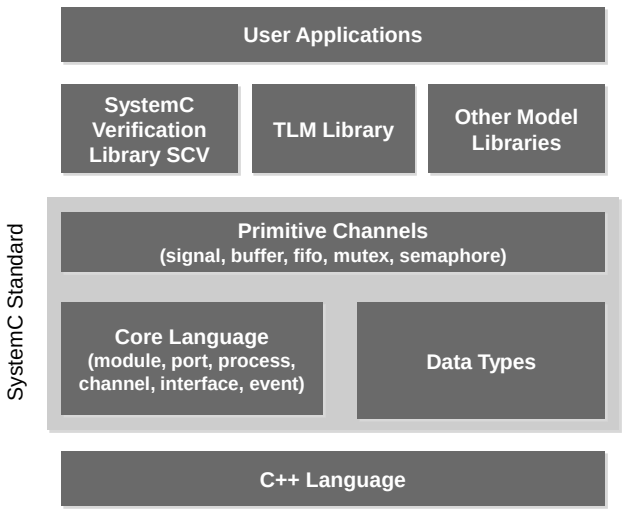
4

Modules and Channels



5

Architecture of SystemC



6

Accellera Systems Initiative

- Formed from Open-SystemC Initiative (OSCI) and Accellera
- Website http://www.accellera.org
- National Users Groups
  - http://www-ti.informatik.uni-tuebingen.de/~systemc
  - http://www.nascug.org
  - http://www.iscug.in
  - Others...
- Language Working Group LWG
  - IEEE 1666™ -2005
  - SystemC 2.2 released Mar 2006 (IEEE 1666 compliant)
  - IEEE 1666™ -2011 and SystemC 2.3

7

Other Working Groups

- Verification Working Group VWG
  - SystemC Verification (SCV) library released 2003
- Synthesis Working Group SWG
  - Synthesisable subset of SystemC
- Transaction-Level Modeling Working Group TLMWG
  - TLM-1.0 released May 2005
  - TLM-2.0 released June 2008
  - TLM-2.0 part of IEEE 1666-2011
- Analog and Mixed Signal Working Group AMSWG

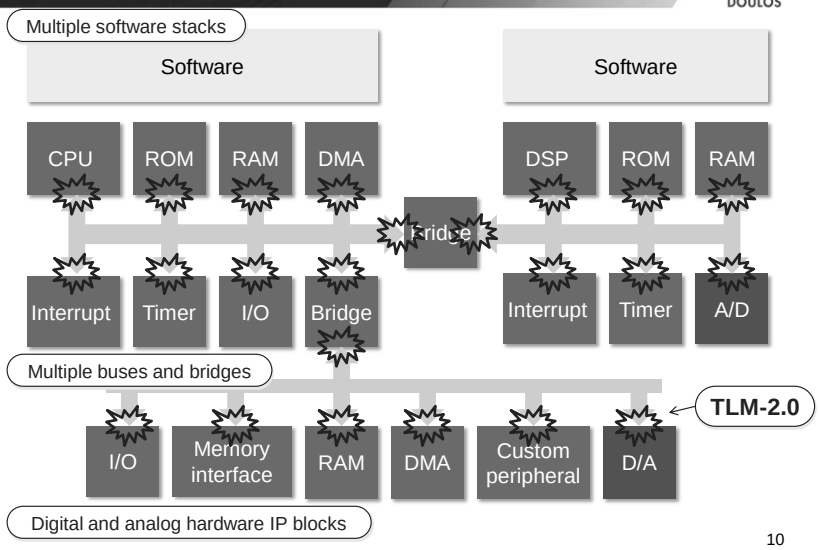
8

What can you do with SystemC?

- Discrete Event Simulation (events, time)
  - Register Transfer Level (delta delays, bus resolution)
  - Transaction Level (communication using function calls)
  - Kahn Process Networks (infinite fifos, reads block when empty)
  - Dataflow (input-execution-output stages)
  - CSP (rendezvous, blocking reads & writes)
- Continuous Time or Frequency Domain (Analog)
  SystemC AMS
- NOT gate level
- NOT abstract RTOS modeling (process scheduling, priorities, preemption)
  (originally planned as version 3)

9

Typical Use Case: Virtual Platform



10

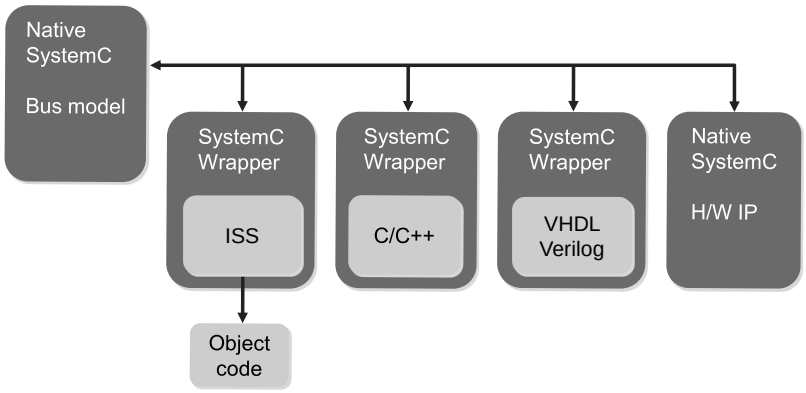
What is SystemC Used For?

- Virtual Platform
  - Architectural exploration, performance modeling
  - Software development
  - Reference model for functional verification
  - Available before RTL - **early!**
  - Simulates much faster than RTL - **fast!**
- High-level synthesis
- Used by
  - Architects, software, hardware, and verification engineers

11

SystemC is Glue!

- Transaction-level modeling is communication-centric



12

Why SystemC in Particular?

- Industry standard IEEE 1666™
- Open-source C++ simulator
  - Available across multiple platforms (Linux, Windows, Mac)
  - No vendor lock-in, no licensing issues
- Easy integration with the C/C++ world
- Easy to put SystemC wrappers around existing HDL or C models
- Wide availability of tools, models and know-how
- Mature tool vendor support for co-simulation and debug

13

Modules, Processes and Ports

- Modules and ports
- Channels and Interfaces
- Processes
- Events

14

SC_MODULE

Class

Ports

Constructor

Process

```
#include "systemc.h"

SC_MODULE(Mult)
{
  sc_in<int>  a;
  sc_in<int>  b;
  sc_out<int> f;

  void action() { f = a * b; }

  SC_CTOR(Mult)
  {
    SC_METHOD(action);
      sensitive << a << b;
  }
};
```

15

Separate Header File

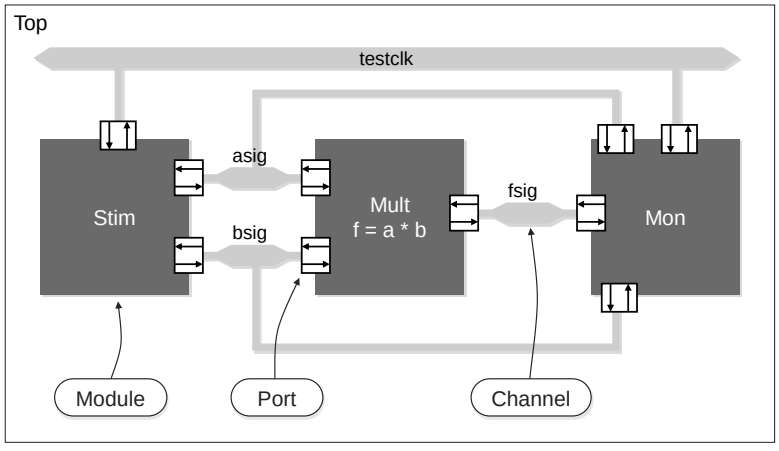
```
// mult.h
#include "systemc.h"

SC_MODULE(Mult)
{
  sc_in<int>  a;
  sc_in<int>  b;
  sc_out<int> f;

  void action();

  SC_CTOR(Mult)
  {
    SC_METHOD(action);
      sensitive << a << b;
  }
};
```

```
// mult.cpp

#include "mult.h"

void Mult::action()
{
  f = a * b;
}
```

- Define constructor in .cpp?
  Yes - explained later

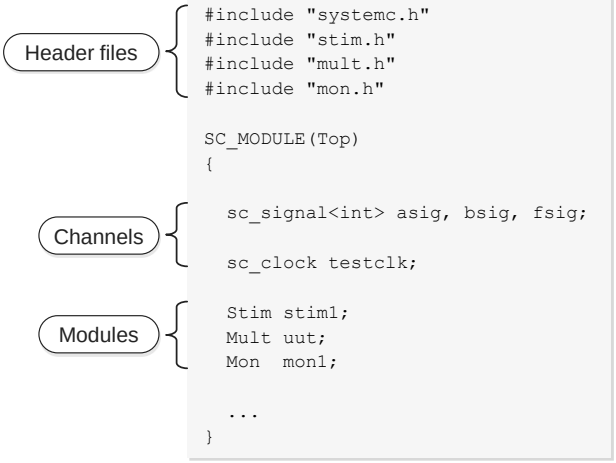
16

The Test Bench



17

Top Level Module




```
#include "systemc.h"
#include "stim.h"
#include "mult.h"
#include "mon.h"

SC_MODULE(Top)
{

  sc_signal<int> asig, bsig, fsig;

  sc_clock testclk;

  Stim stim1;
  Mult uut;
  Mon  mon1;

  ...
}
```

18

Module Instantiation

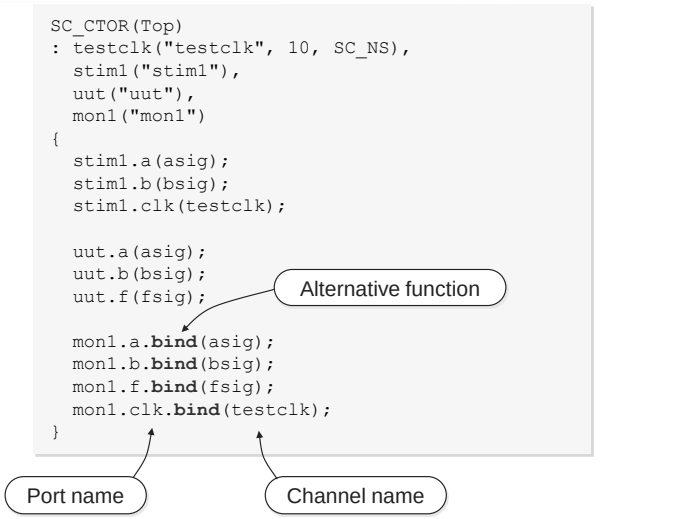
```
SC_MODULE(Top)
{
  sc_signal<int> asig, bsig, fsig;

  sc_clock testclk;

  Stim stim1;
  Mult uut;
  Mon  mon1;

  SC_CTOR(Top)
  : testclk("testclk", 10, SC_NS),
    stim1("stim1"),
    uut  ("uut"),
    mon1 ("mon1")
  {
    ...
  }
}
```

Name of data member

String name of instance (constructor argument)

19

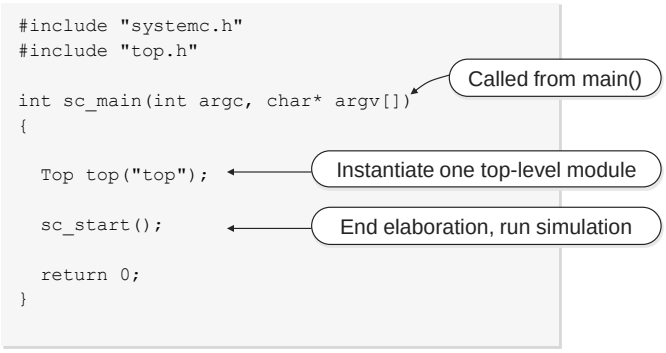
Port Binding



20

sc_main

- sc_main is the entry point to a SystemC application



21

An Alternative to sc_main

- A SystemC implementation is not obliged to support sc_main

```
#include "top.h"

SC_MODULE_EXPORT(Top);

// NCSC_MODULE_EXPORT(Top);
```

QuestaSim

Incisive

Tool-specific macro

22

Namespaces

```
#include "systemc.h"

SC_MODULE(Mod)
{
  sc_in<bool> clk;
  sc_out<int> out;

  ... cout << endl;
```

Old header - global namespace

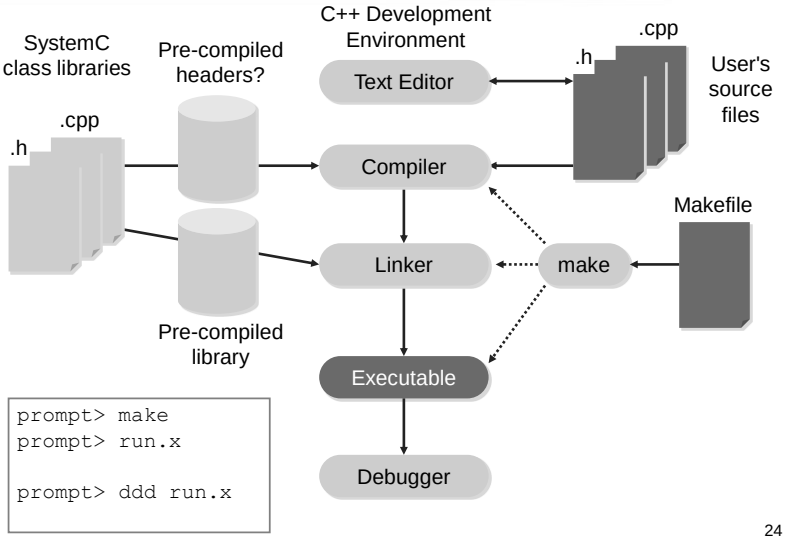
```
#include "systemc"

SC_MODULE(Mod)
{
  sc_core::sc_in<bool> clk;
  sc_core::sc_out<int> out;

  ... std::cout << std::endl;
```

New header

```
#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using std::cout;
using std::endl;

SC_MODULE(Mod) {
  sc_in<bool> clk;
  sc_out<int> out;
  ... cout << endl;
```

23

Compilation and Simulation



24

Kinds of Process

- Processes
  - Must be within a module (not in a function)
  - A module may contain many processes
- Three different kinds of process
  - Methods `SC_METHOD`
  - Threads `SC_THREAD`
  - Clocked threads `SC_CTHREAD` (for synthesis)
- Processes can be *static* or *dynamic*

25

SC_METHOD Example

```
#include <systemc.h>

template<class T>
SC_MODULE(Register)
{
  sc_in<bool> clk, reset;
  sc_in<T>    d;
  sc_out<T>   q;

  void entry();

  SC_CTOR(Register)
  {
    SC_METHOD(entry);
      sensitive << reset;
      sensitive << clk.pos();
  }
};
```

```
template<class T>
void Register<T>::entry()
{
  if (reset)
    q = 0; // promotion
  else if (clk.posedge())
    q = d;
}
```

- SC_METHODs execute in zero time
- SC_METHODs cannot be suspended
- SC_METHODs should not contain infinite loops

26

SC_THREAD Example

```
#include "systemc.h"

SC_MODULE(Stim)
{
  sc_in<bool> Clk;
  sc_out<int> A;
  sc_out<int> B;

  void stimulus();

  SC_CTOR(Stim)
  {
    SC_THREAD(stimulus);
      sensitive << Clk.pos();
  }
};
```

```
#include "stim.h"

void Stim::stimulus()
{
  wait();
  A = 100;
  B = 200;
  wait();      // for Clk edge
  A = -10;
  B = 23;
  wait();
  A = 25;
  B = -3;
  wait();
  sc_stop();   // Stop simulation
}
```

- More general and powerful than an `SC_METHOD`
- Simulation typically slower than an `SC_METHOD`
- Called once only: hence often contains an infinite loop

27

SC_HAS_PROCESS

```
#include "systemc.h"
class Counter: public sc_module
{
public:
  sc_in<bool> clock, reset;
  sc_out<int> q;

  Counter(sc_module_name _nm, int _mod)
  : sc_module(_nm), count(0) , modulus(_mod)
  {
    SC_METHOD(do_count);
      sensitive << clock.pos();
  }

  SC_HAS_PROCESS(Counter);

private:
  void do_count();
  int count;
  const int modulus;
};
```

Constructor arguments

Needed if there's a process and not using SC_CTOR

28

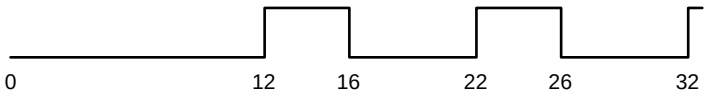
sc_clock

sc_time sc_time

```
sc_clock clk("clk",    period, 0.4, firstedge, true);

sc_clock clk("clk", 10, SC_NS, 0.4, 12, SC_NS, true);
```

Duty cycle

1st edge rising




```
("clock_N", 1, SC_NS, 0.5,  0, SC_NS, true);
```

Defaults

Avoid clocks altogether for fast models!

29

Dynamic Sensitivity

```
SC_CTOR(Module)
{
  SC_THREAD(thread);
    sensitive << a << b;
}

void thread()
{
  for (;;)
  {
    wait();
    ...
    wait(10, SC_NS);
    ...
    wait(e);
    ...
  }
}
```

Static sensitivity list

Wait for event on *a* or *b*

Wait for 10ns

Wait for event *e*

ignore *a* or *b*

30

sc_event and Synchronisation

```
SC_MODULE(Test)
{
  int data;
  sc_event e;
  SC_CTOR(Test)
  {
    SC_THREAD(producer);
    SC_THREAD(consumer);
  }
  void producer()
  {
    wait(1, SC_NS);
    for (data = 0; data < 10; data++) {
      e.notify();
      wait(1, SC_NS);
    }
  }
  void consumer()
  {
    for (;;) {
      wait(e);
      cout << "Received " << data << endl;
    }
  }
};
```

Shared variable

Schedule event immediately

Resume when event occurs

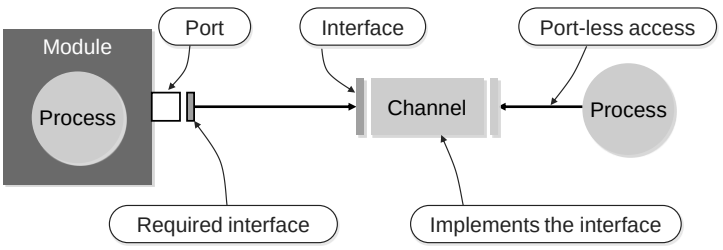
31

Kinds of Channel

- Primitive channels
  - Implement one or more interfaces
  - Derived from sc_prim_channel
  - Have access to the update phase of the scheduler
  - Examples - sc_signal, sc_signal_resolved, sc_fifo
- Hierarchical channels
  - Implement one or more interfaces
  - Derived from sc_module
  - Can instantiate ports, processes and modules
- Minimal channels - implement one or more interfaces

32

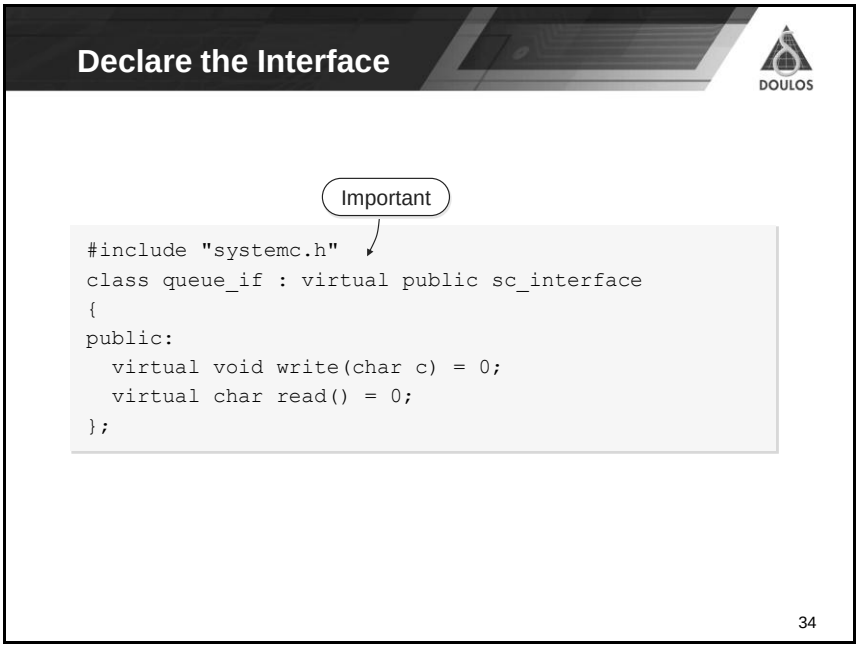
Interface Method Call



- An interface declares of a set of methods (pure virtual functions)
- An interface is an abstract base class of the channel
- A channel *implements* an interface (c.f. Java)
- A module calls interface methods via a port

33

Declare the Interface



34

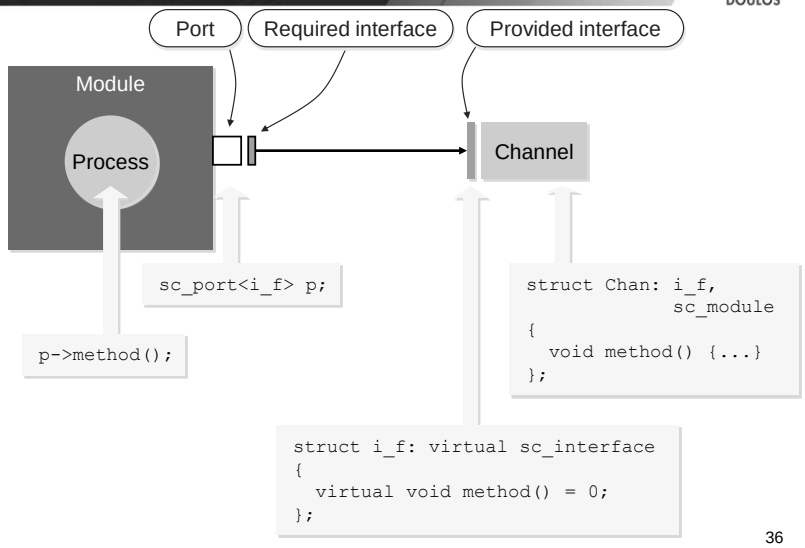
Queue Channel Implementation

```
#include "queue_if.h"
class Queue : public queue_if, public sc_object
{
public:
  Queue(char* nm, int _sz)
  : sc_object(nm), sz(_sz)
  { data = new char[sz]; w = r = n = 0; }

  void write(char c);
  char read();

private:
  char* data;
  int sz, w, r, n;
};
```

Implements interface methods

35

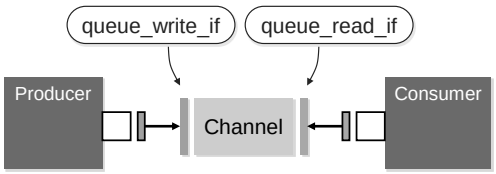
Understanding Ports



36

Queue Ports

```
class Producer : public sc_module
{
public:
  sc_port<queue_write_if> out;

  void do_writes();

  SC_CTOR(Producer)
  {
    SC_THREAD(do_writes);
  }
};
```



37

Calling Methods via Ports

```
#include "systemc.h"
#include "producer.h"

void Producer::do_writes()
{
  std::string txt = "Hallo World.";
  for (int i = 0; i < txt.size(); i++)
  {
    wait(SC_ZERO_TIME);

    out->write(txt[i]);
  }
}
```

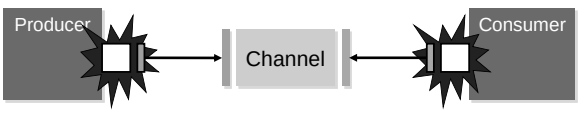
Note: -> overloaded

Interface Method Call

38

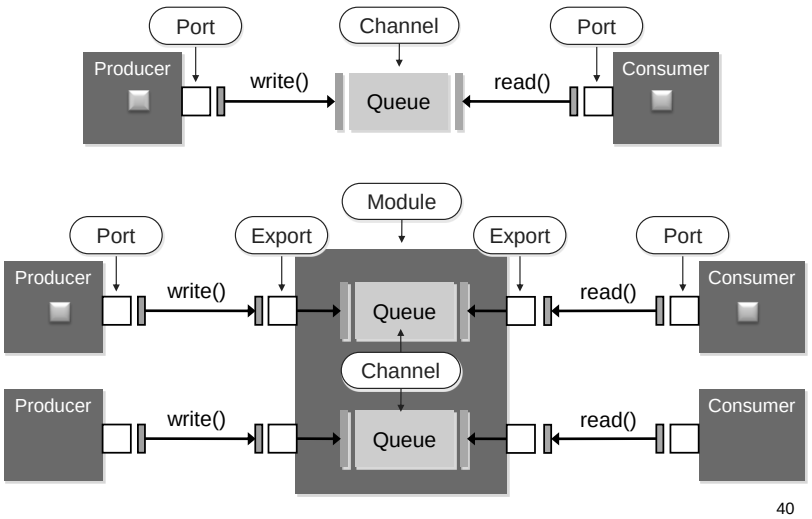
Why Ports?

- Ports allow modules to be independent of their environment
- Ports support elaboration-time checks (register_port, end_of_elaboration)
- Ports can have data members and member functions



39

Exports



40

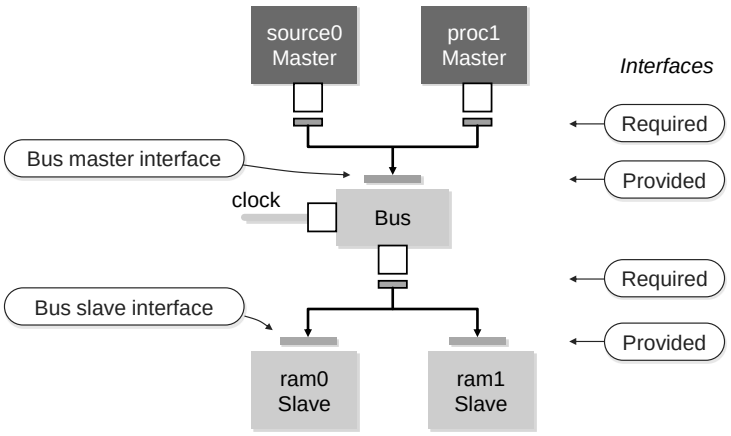
Bus Modeling

- Master and slave interfaces
- Blocking versus non-blocking
- Multiports

41

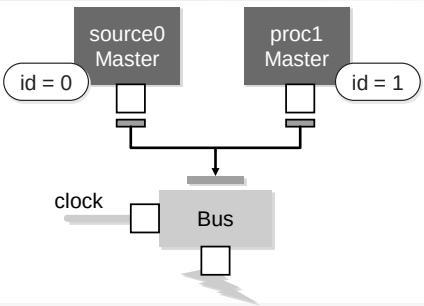
Example Bus Model

- Multiple bus masters (modules), shared bus (channel), multiple slaves (channels)
- Bus arbitration and memory mapping built into the bus



42

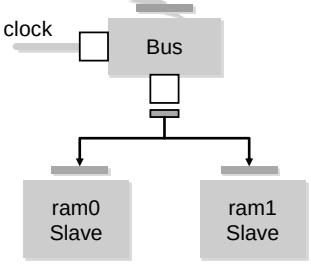
Master Interface Definition




```
class master_if : virtual public sc_interface
{
public:
  virtual void write(sc_uint<8> address, sc_uint<12> data,
                     int id) = 0;
  virtual void read (sc_uint<8> address, sc_uint<12> &data,
                     int id) = 0;
};
```

43

Slave Interface Definition

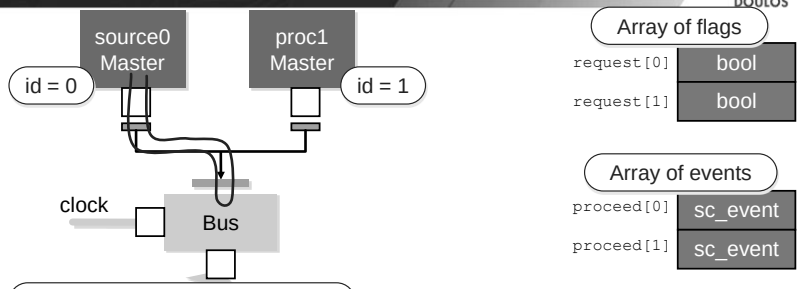



```
class slave_if : virtual public sc_interface
{
public:
  virtual void slave_write(sc_uint<8> address, sc_uint<12>  data) = 0;
  virtual void slave_read (sc_uint<8> address, sc_uint<12> &data) = 0;
  virtual void get_map(unsigned int &start, unsigned int &size)   = 0;
};
```

Memory map managed within bus channel

44

Master Write and Read



Runs in the context of the caller

```
void Bus::write(sc_uint<8> address, sc_uint<12> data, int id)
{
  request[id] = true;         // request access to the bus
  wait(proceed[id]);          // wait for permission
  request[id] = false;        // clear the flag
  slave_port[find_port(address)]->slave_write(address, data);
}
```

Blocking and Non-blocking Calls

- An Interface Method Call runs in the context of the caller

Important!

- OSCI terminology:
  - A *blocking* method may call wait
  - A *blocking* method must be called from a thread process
  - A *non-blocking* method must not call wait
  - A *non-blocking* method may be called from a thread or method process
  - Naming convention nb_*

46

Bus Controller Process

```
void Bus::control_bus()
{
  int highest;
  for (;;)
  {
    wait();

    // Pick out a master that's made a request
    highest = -1;
    for (int i = 0; i < n_masters; i++)
      if (request[i])
        highest = i;

    // Notify the master with the highest id
    if (highest > -1)
      proceed[highest].notify();
  }
}
```

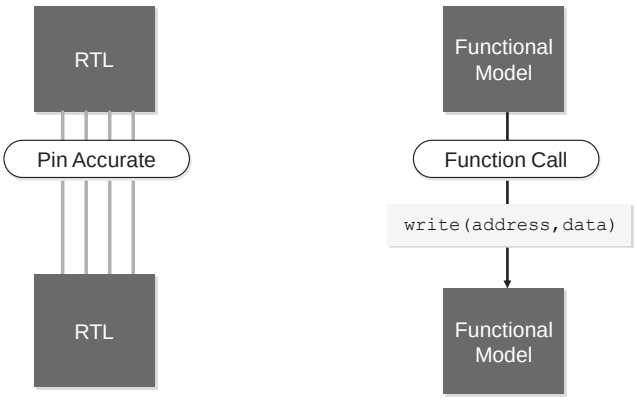
47

Introduction to TLM-2.0

- Transaction Level Modeling
- The architecture of TLM-2.0
- Initiator, interconnect, and target
- Sockets
- The generic payload
- Loosely-timed coding style
- DMI

48

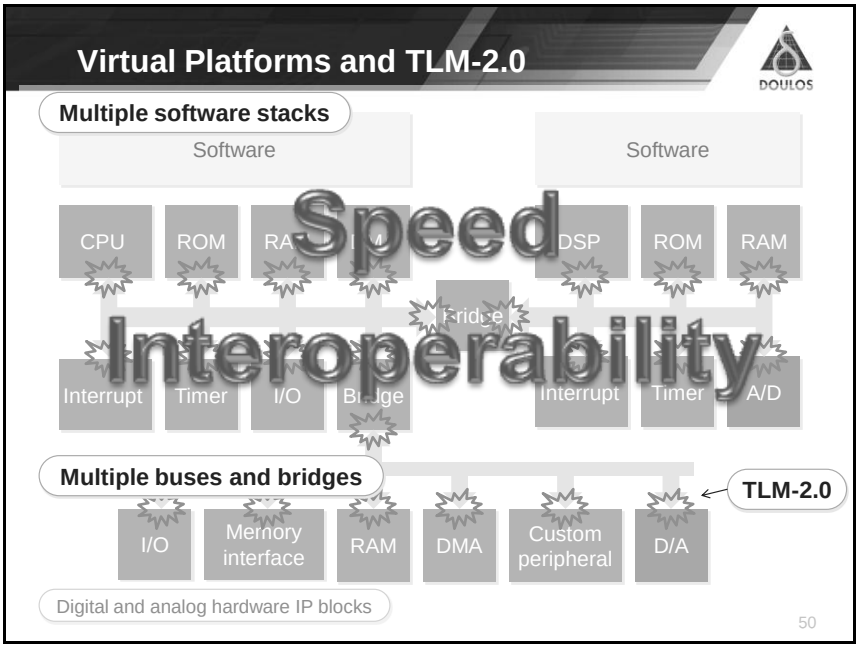
Transaction Level Modeling

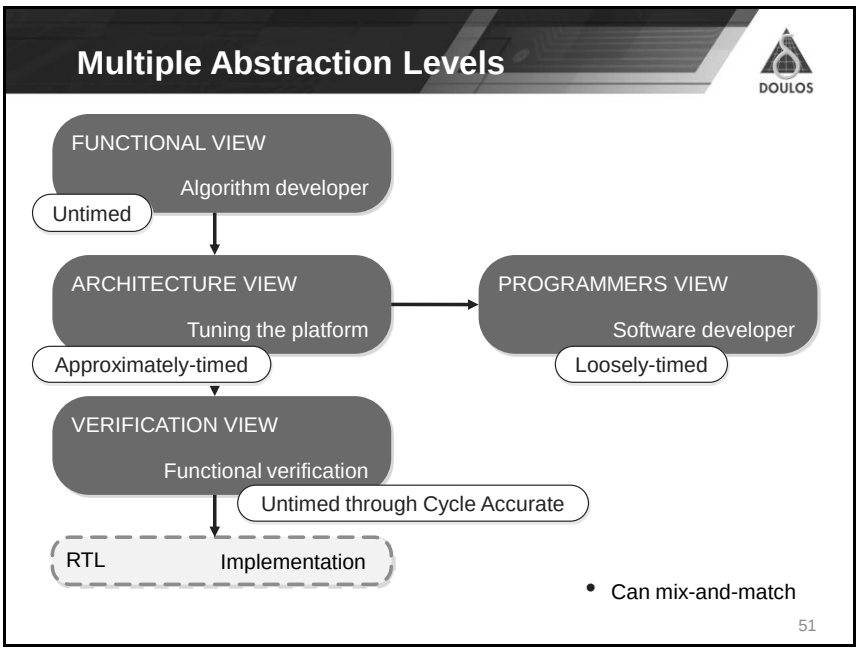


Simulate every event!

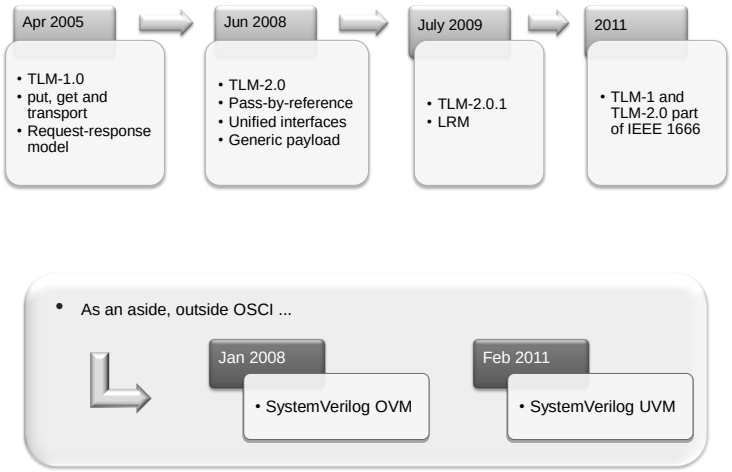
100-10,000 X faster simulation!

49



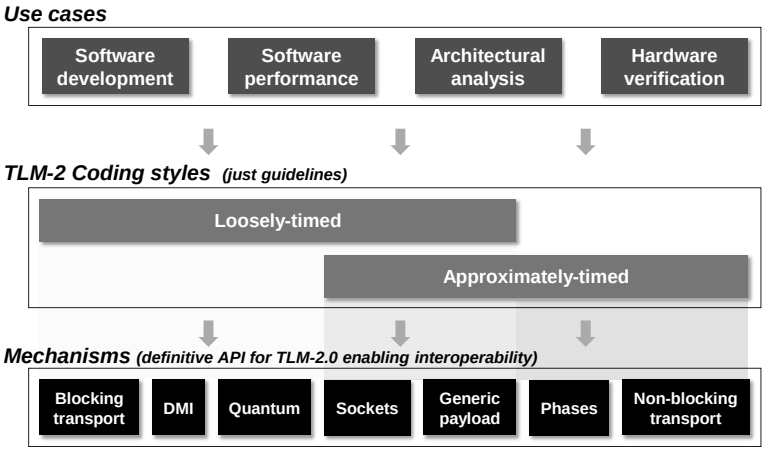


OSCI TLM Development



52

Use Cases, Coding Styles and Mechanism



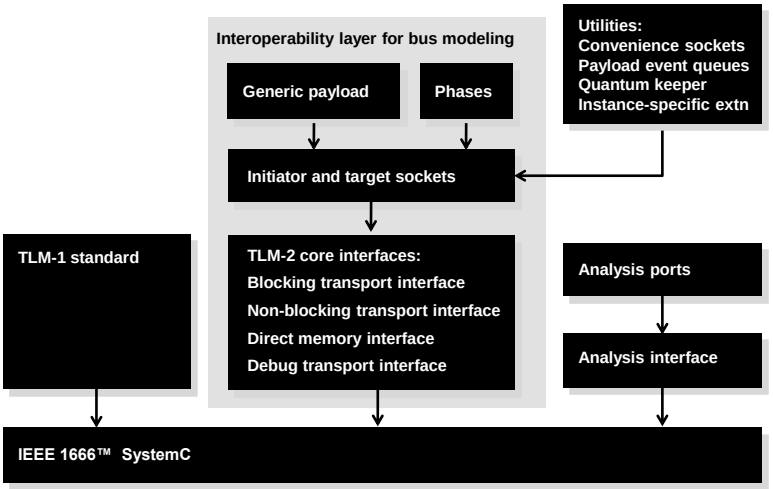
53

Coding Styles

- Loosely-timed = as fast as possible
  - Register-accurate
  - Only sufficient timing detail to boot O/S and run multi-core systems
  - b_transport – each transaction completes in one function call
  - Temporal decoupling
  - Direct memory interface (DMI)
- Approximately-timed = just accurate enough for performance modeling
  - *aka* cycle-approximate or cycle-count-accurate
  - Sufficient for architectural exploration
  - nb_transport – each transaction has 4 timing points (extensible)
- Guidelines only – not definitive

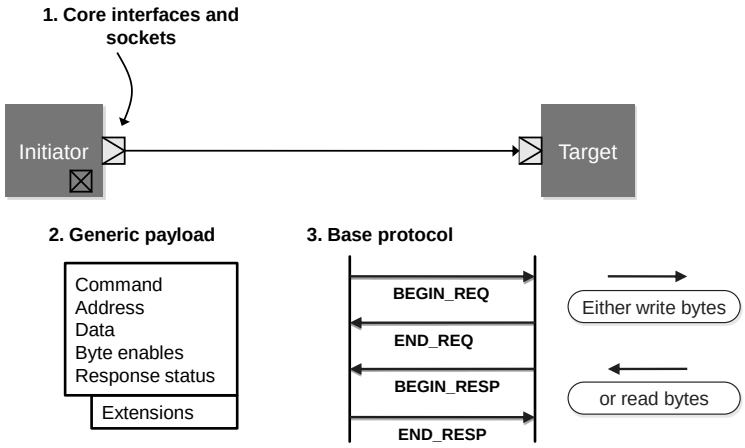
54

The TLM 2.0 Classes



55

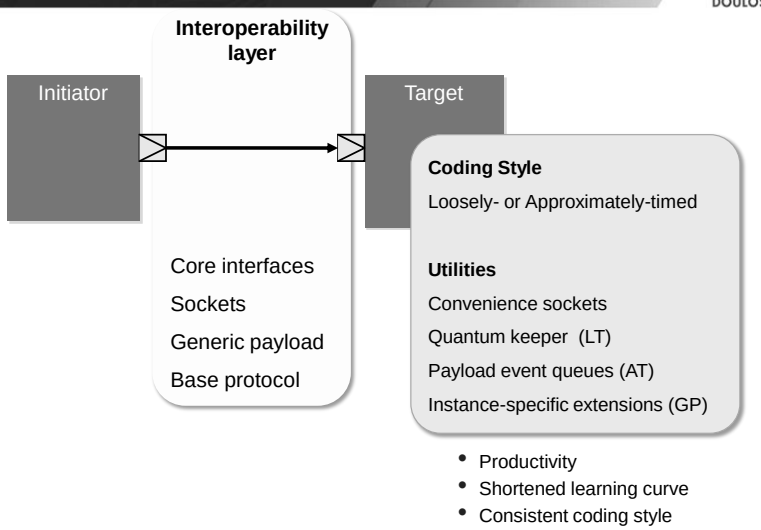
Interoperability Layer



- Maximal interoperability for memory-mapped bus models

56

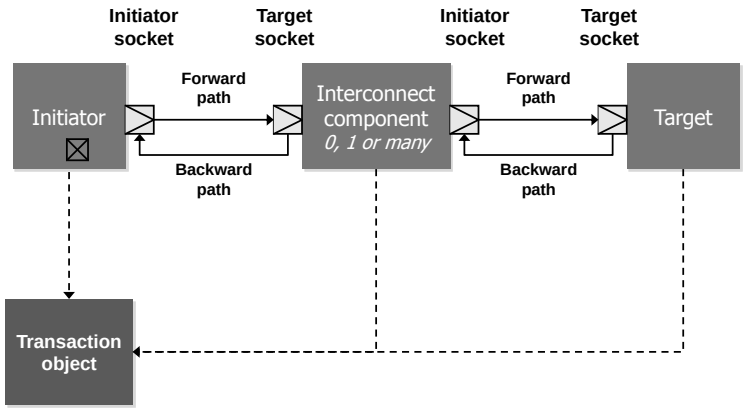
Utilities



- Productivity
- Shortened learning curve
- Consistent coding style

57

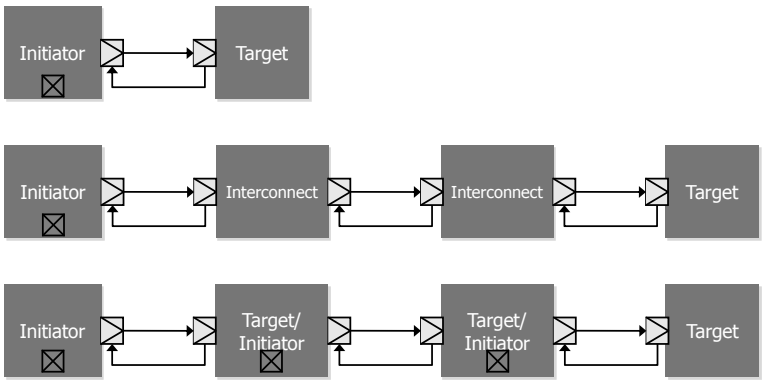
Initiators, Targets and Interconnect



- Single transaction object for request and response
- References to the object are passed along the forward and backward paths

58

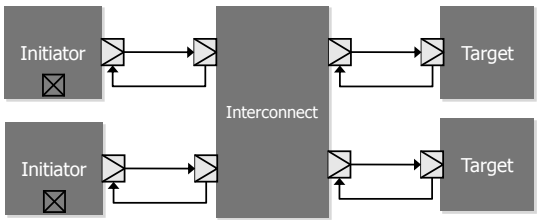
TLM-2 Connectivity



- Roles are dynamic; a component can choose whether to act as interconnect or target
- Transaction memory management needed

59

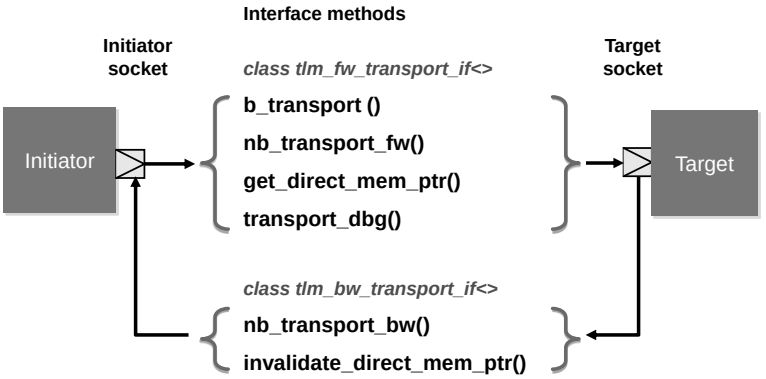
Convergent Paths



- Paths not predefined; routing may depend on transaction attributes (e.g. address)
- Whether arbitration is needed depends on the coding style

60

Initiator and Target Sockets



- Sockets provide fw and bw paths, and group interfaces

61

Initiator Socket

```
#include "tlm.h"
struct  Initiator: sc_module,  tlm::tlm_bw_transport_if<>
{
   tlm::tlm_initiator_socket<>  init_socket;

   SC_CTOR(Initiator) : init_socket("init_socket") {
      SC_THREAD(thread);
      init_socket.bind( *this );
   }

   void thread() { ...
      init_socket->b_transport( trans, delay );
      init_socket->nb_transport_fw( trans, phase, delay );
      init_socket->get_direct_mem_ptr( trans, dmi_data );
      init_socket->transport_dbg( trans );
   }

   virtual  tlm::tlm_sync_enum  nb_transport_bw( ... ) { ... }
   virtual void  invalidate_direct_mem_ptr( ... ) { ... }
};
```

Combined interface required by socket

Protocol type defaults to base protocol

Initiator socket bound to initiator itself

Calls on forward path

Methods for backward path

tlm_initiator_socket must be bound to object that implements entire bw interface

62

Target Socket

```
struct  Target: sc_module,  tlm::tlm_fw_transport_if<>
{
   tlm::tlm_target_socket<>  targ_socket;

   SC_CTOR(Target) : targ_socket("targ_socket") {
      targ_socket.bind( *this );
   }
   virual void b_transport( ... ) { ... }
   virtual tlm::tlm_sync_enum  nb_transport_fw( ... ) { ... }
   virtual bool get_direct_mem_ptr( ... ) { ... }
   virtual unsigned int transport_dbg( ... ) { ... }
};
```

Combined interface required by socket

Protocol type default to base protocol

Target socket bound to target itself

Methods for forward path

tlm_target_socket must be bound to object that implements entire fw interface

63

Socket Binding


```
SC_MODULE(Top)  {
   Initiator  *init;
   Target    *targ;

   SC_CTOR(Top)  {
      init  = new Initiator("init");
      targ = new Target("targ");
      init->init_socket.bind( targ->targ_socket );
   }
   ...
```

Bind initiator socket to target socket

64

The Generic Payload

- Has typical attributes of a memory-mapped bus

Attribute	Type	Modifiable?
Command	tlm_command	No
Address	uint64	Interconnect only
Data pointer	unsigned char*	No (array – yes)
Data length	unsigned int	No
Byte enable pointer	unsigned char*	No
Byte enable length	unsigned int	No
Streaming width	unsigned int	No
DMI hint	bool	Yes
Response status	tlm_response_status	Target only
Extensions	(tlm_extension_base*)[ ]	Yes

65

Response Status

enum tlm_response_status	Meaning
TLM_OK_RESPONSE	Successful
TLM_INCOMPLETE_RESPONSE	Transaction not delivered to target (default)
TLM_ADDRESS_ERROR_RESPONSE	Unable to act on address
TLM_COMMAND_ERROR_RESPONSE	Unable to execute command
TLM_BURST_ERROR_RESPONSE	Unable to act on data length/ streaming width
TLM_BYTE_ENABLE_ERROR_RESPONSE	Unable to act on byte enable
TLM_GENERIC_ERROR_RESPONSE	Any other error

- Set to TLM_INCOMPLETE_RESPONSE by the initiator
- May be modified by the target
- Checked by the initiator when transaction is complete

66

Generic Payload - Initiator

```
void thread_process() {  // The initiator
    tlm::tlm_generic_payload  trans;                    Would usually pool transactions
    sc_time  delay = SC_ZERO_TIME;

    trans.set_command( tlm::TLM_WRITE_COMMAND );        8 attributes you must set
    trans.set_data_length( 4 );
    trans.set_streaming_width( 4 );
    trans.set_byte_enable_ptr( 0 );

    for ( int i = 0; i < RUN_LENGTH; i += 4 ) {
        int  word = i;
        trans.set_address( i );
        trans.set_data_ptr( (unsigned char*)( &word ) );
        trans.set_dmi_allowed( false );
        trans.set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

        init_socket->b_transport( trans, delay );

        if ( trans.is_response_error() )
            SC_REPORT_ERROR("TLM-2", trans.get_response_string().c_str());
        ...
}
```

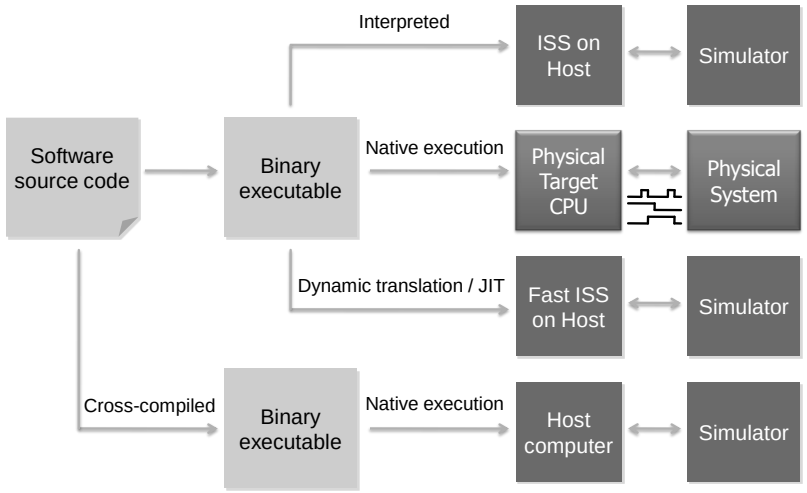
67

Generic Payload - Target

```
virtual  void  b_transport(  // The target
    tlm::tlm_generic_payload& trans,  sc_core::sc_time& t )
{
    tlm::tlm_command   cmd   = trans.get_command();          // 6 attributes you must check
    sc_dt::uint64      adr   = trans.get_address();
    unsigned char*     ptr   = trans.get_data_ptr();
    unsigned int       len   = trans.get_data_length();
    unsigned char*     byt   = trans.get_byte_enable_ptr();
    unsigned int       wid   = trans.get_streaming_width();

    if ( byt != 0 || len > 4 || wid < len || adr+len > memsize ) {   // Target supports 1-word transfers
        trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
        return;
    }

    if ( cmd == tlm::TLM_WRITE_COMMAND )                    // Execute command
        memcpy( &m_storage[adr], ptr, len );
    else if ( cmd == tlm::TLM_READ_COMMAND )
        memcpy( ptr, &m_storage[adr], len );

    trans.set_response_status( tlm::TLM_OK_RESPONSE );      // Successful completion
}
```

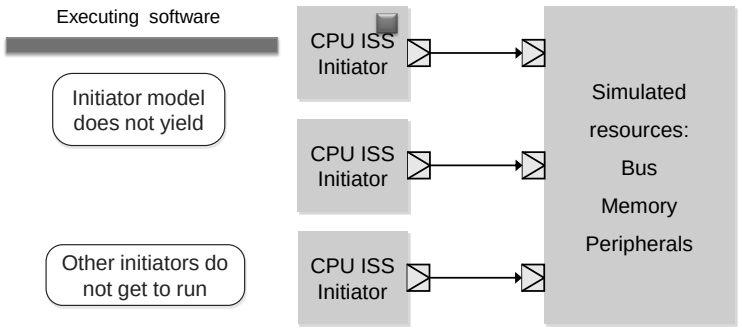
68

Software Execution and Simulation



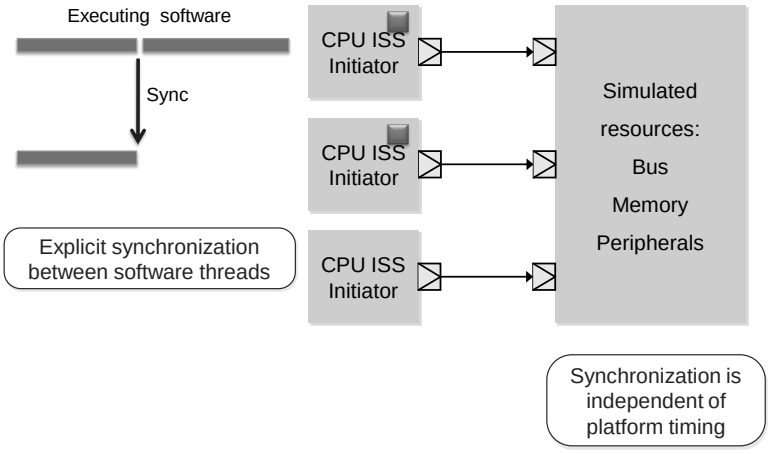
69

Software execution without Sync



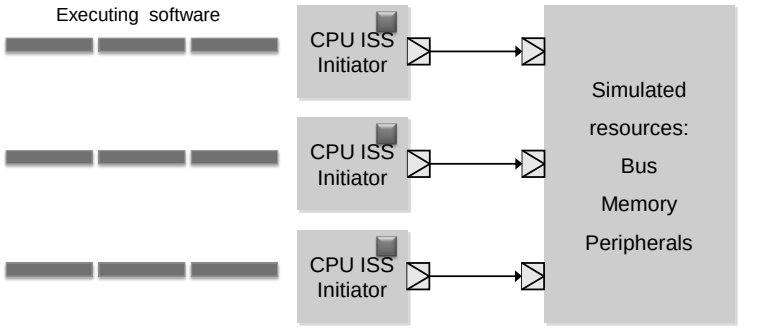
70

Software execution with Sync



71

Software execution with a Quantum



Initiators use a time quantum

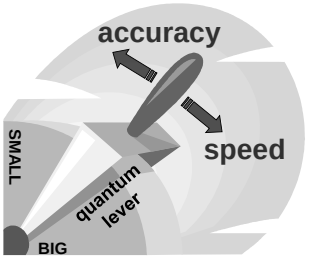
Each initiator gets a time slice

Resources consume simulation time

72

The Quantum

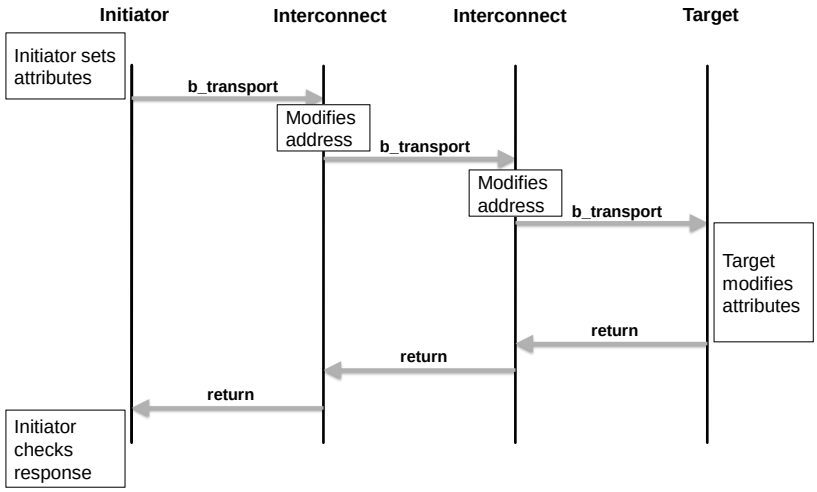
- Quantum can be set by user



- Typically, all initiators use the same global quantum
- Individual initiators can be permitted to sync more frequently
- Not obliged to use the quantum at all if there are explicit synchronization points
- Quantum could be changed dynamically to “zoom in” (but no explicit support)

73

Causality with b_transport



74

Timing Annotation and b_transport

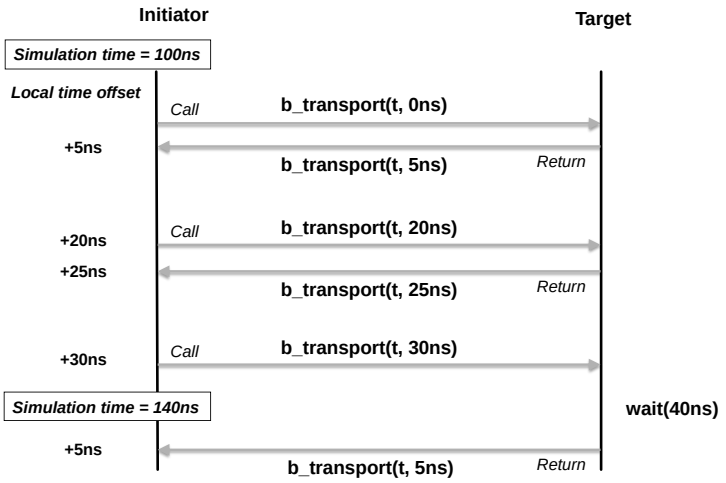
```
virtual  void  b_transport ( TRANS& trans , sc_core::sc_time& delay )
{
  Behave as if method were called at sc_time_stamp() + delay
  ...
  delay = delay + latency;
}
```

```
socket->b_transport( transaction, delay );

Behave as if method returned at sc_time_stamp() + delay
```

- Recipient may
  - Execute transactions immediately, out-of-order – Loosely-timed
  - Schedule transactions to execute at proper time – Approx-timed
  - Pass on the transaction with the timing annotation

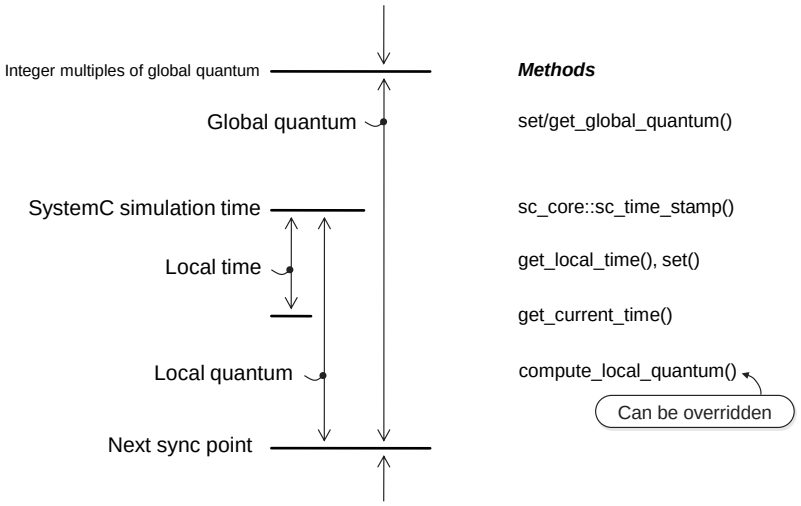
75

Temporal Decoupling



76

Quantum Keeper Variables



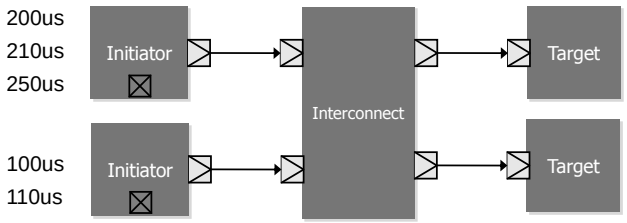
77

Quantum Keeper Example

```
#include "tlm_utils/tlm_quantumkeeper.h"
struct Initiator: sc_module   {
   tlm::tlm_initiator_socket<>  init_socket;
   tlm_utils::tlm_quantumkeeper  m_qk;                    // The quantum keeper
   SC_CTOR(Initiator) : init_socket("init_socket")  {
      m_qk.set_global_quantum( sc_time(1, SC_US) );       // Replace the global quantum
      m_qk.reset();                                       // Recalculate the local quantum
   }
   void thread() { ...
      for (int i = 0; i < RUN_LENGTH; i += 4) {
         ...
         delay = m_qk.get_local_time() ;
         init_socket->b_transport( trans, delay );
         m_qk.set_and_sync( delay );                      // Check local time against quantum
                                                          // and sync if necessary
```

78

Base Protocol Rules



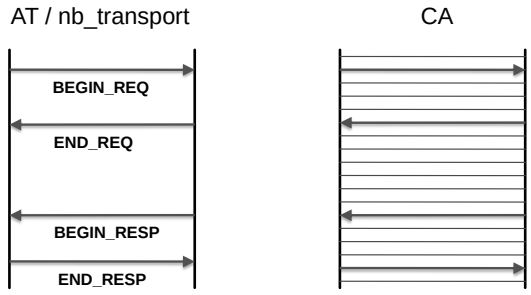
- Each initiator should generate transactions in non-decreasing time order
- Targets usually return immediately (don't want b_transport to block)
- b_transport is re-entrant anyway
- Incoming transactions through different sockets may be out-of-order
- Out-or-order transactions can be executed in any order
- Arbitration is typically inappropriate (and too slow)

Custom protocols make their own rules

79

AT and CA

- No running ahead of simulation time; everything stays in sync

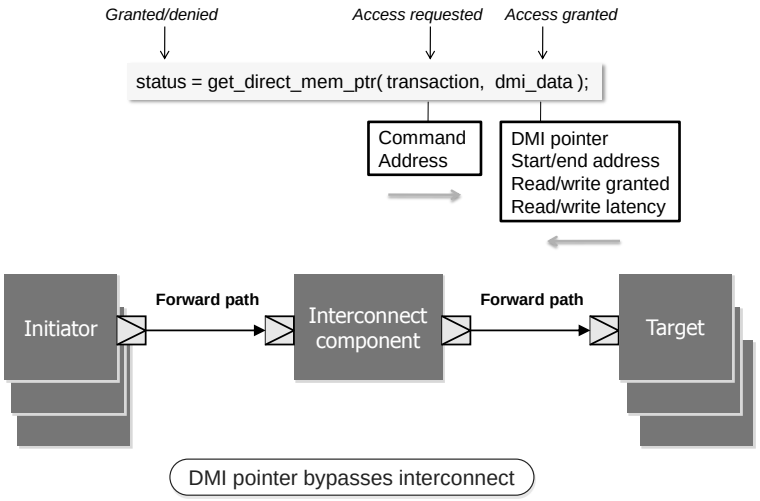


Wake up at significant timing points

Wake up every cycle

80

Direct Memory Interface



81

DMI Example 1

```
struct Initiator: sc_module
{
  bool  dmi_ptr_valid;                                  Returned by get_direct_mem_ptr
  tlm::tlm_dmi  dmi_data;

  void thread_process()
  {
    if (dmi_ptr_valid  &&  a >= dmi_data.get_start_address()      Try existing DMI region
                       &&  a <= dmi_data.get_end_address() )
    {
      if ( cmd == tlm::TLM_READ_COMMAND && dmi_data.is_read_allowed() )
      {
        memcpy(&data, dmi_data.get_dmi_ptr() + a - dmi_data.get_start_address(), 4);
        local_time += dmi_data.get_read_latency();
      }
      else if ( cmd == tlm::TLM_WRITE_COMMAND && dmi_data.is_write_allowed() )
      {
        memcpy(dmi_data.get_dmi_ptr() + a - dmi_data.get_start_address(), &data, 4);
        local_time += dmi_data.get_write_latency();
      }
      else
        ...
```

82

DMI Example 2

```
  ...
  else {                                                DMI is invalid
    trans->set_address( a );                            Setup regular transaction
    ...
    socket->b_transport( *trans, delay  );              May modify address attribute
    if ( trans->is_response_error() )
      ...

    if ( trans->is_dmi_allowed() )                      Test DMI hint
    {
      trans->set_address( a );                          Reuse transaction object
      dmi_data.init();                                  Reset DMI descriptor
      dmi_ptr_valid = socket->get_direct_mem_ptr( *trans, dmi_data );
    }
    ...
```

83

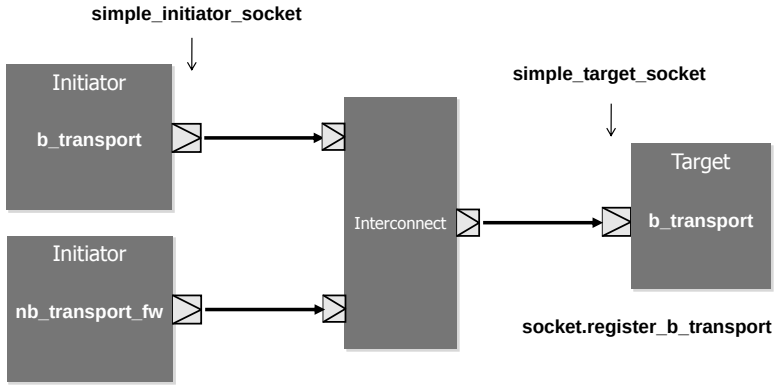
Simple Sockets

```
struct  Interconnect : sc_module
{
  tlm_utils::simple_target_socket<Interconnect>    targ_socket;
  tlm_utils::simple_initiator_socket<Interconnect> init_socket;

  SC_CTOR(Interconnect) : targ_socket("targ_socket"), init_socket("init_socket")
  {
    targ_socket.register_nb_transport_fw(    this, &Interconnect::nb_transport_fw);
    targ_socket.register_b_transport(        this, &Interconnect::b_transport);
    targ_socket.register_get_direct_mem_ptr(this, &Interconnect::get_direct_mem_ptr);
    targ_socket.register_transport_dbg(      this, &Interconnect::transport_dbg);

    init_socket. register_nb_transport_bw(   this, &Interconnect::nb_transport_bw);
    init_socket. register_invalidate_direct_mem_ptr(
                                             this,  &Interconnect::invalidate_direct_mem_ptr);
  }
  virtual void                  b_transport( ... );
  virtual tlm::tlm_sync_enum    nb_transport_fw( ... );
  virtual bool                  get_direct_mem_ptr( ... );
  virtual unsigned int          transport_dbg( ... );
  virtual tlm::tlm_sync_enum    nb_transport_bw( ... );
  virtual void                  invalidate_direct_mem_ptr( ...);
};
```

Only register the methods you use

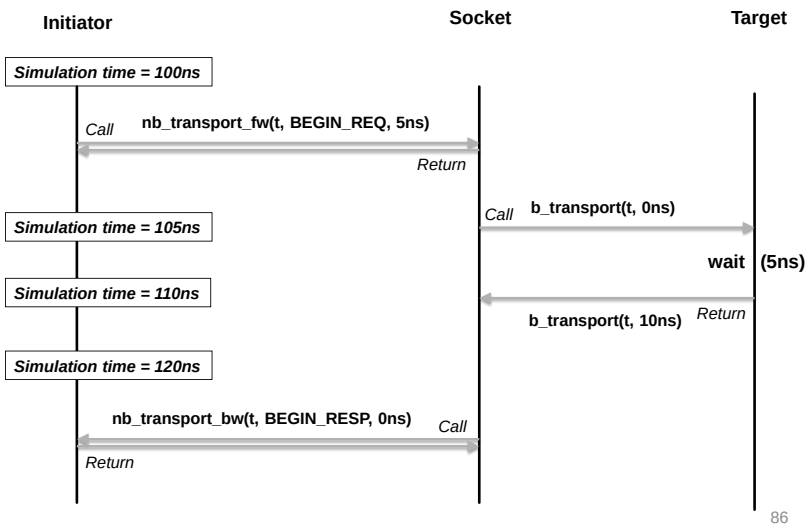
84

b/nb Conversion



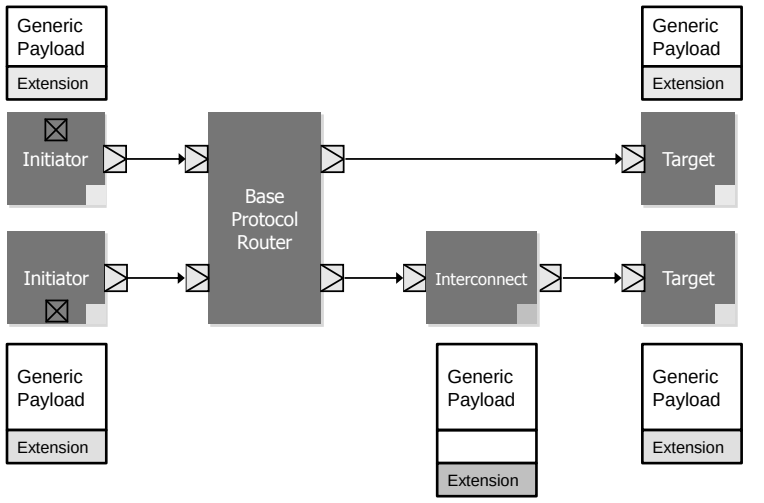
85

Simple Target Socket nb/b Adapter



86

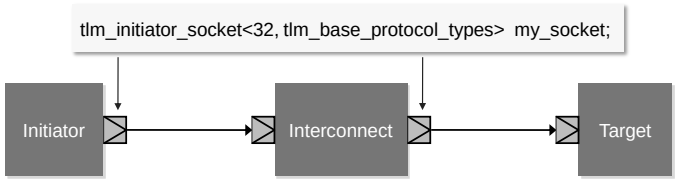
Extensions



87

First Kind of Interoperability

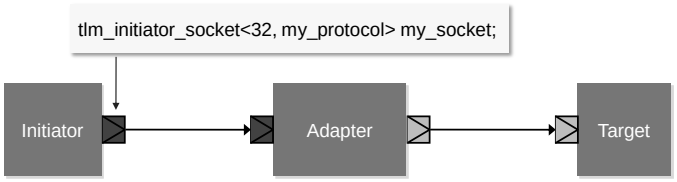
- Use the full interoperability layer
- Use the generic payload + ignorable extensions as an abstract bus model
- Obey all the rules of the base protocol. The LRM is your rule book



88

Second Kind of Interoperability

- Create a new protocol traits class
- Create user-defined generic payload extensions and phases as needed
- Make your own rules!



- One rule enforced: cannot bind sockets of differing protocol types
- Recommendation: keep close to the base protocol. The LRM is your guidebook
- The clever stuff in TLM-2.0 makes the adapter fast

89

For More FREE Information

- IEEE 1666

standards.ieee.org/getieee/1666/download/1666-2011.pdf

- OSCI SystemC 2.2

www.accellera.org

- On-line tutorials

www.doulos.com/knowhow/systemc

90