Virtual Prototyping using SystemC and TLM-2.0

John Aynsley, Doulos
Virtual Prototyping using SystemC and TLM-2.0

• Introduction
• Development of TLM-2.0
• TLM-2.0 Sockets and Interfaces
• LT, AT, and CA
• Generic Payload and Extensions
• Interoperability
What is SystemC?

- System-level modeling language
  - Network of communicating processes (c.f. HDL)
  - Supports heterogeneous models-of-computation
  - Models hardware and software, digital and analog

- C++ class library
  - Open source proof-of-concept simulator
  - Owned by Accellera (previously OSCI)
Architecture of SystemC

User Applications

TLM-1 and TLM-2.0

SystemC Verification Library SCV

SystemC-AMS

Primitive Channels
(signal, buffer, fifo, mutex, semaphore)

Core Language
(module, port, process, channel, interface, event)

Data Types

C++ Language

IEEE 1666 Standard
TLM Standardization

Apr 2005
• TLM-1.0
  • Unidirectional message-passing interfaces

Jun 2008
• TLM-2.0
  • Pass-by-reference
  • Unified interfaces
  • Generic payload

July 2009
• LRM
  • TLM-2.0.1

2011
• TLM-1 and TLM-2.0 part of IEEE 1666
  • TLM-2.0.2

Jan 2008
• SystemVerilog OVM - UVM

Beyond SystemC...
© Accellera Systems Initiative
What can you do with SystemC?

• Discrete Event Simulation  (events, time)
  • Register Transfer Level  (delta delays, bus resolution)
  • Behavioral Modeling  (functions, processes, parallelism)
  • 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)

• Analog at ESL  (SystemC AMS)
• High-Level Synthesis  (synthesis subset)

• NOT gate level
• NOT Spice level
• NOT software / RTOS modeling  (process scheduling, priorities, preemption)
What is SystemC Used For?

• Behavioral Modeling and Reference Models

• Virtual Platforms (aka Software Virtual Prototypes)
  • Architectural exploration, performance modeling
  • Software development
  • Reference model for functional verification

• High-Level Synthesis (C/C++)
Transaction-Level Modeling

Simulate every event! 100-10,000 X faster simulation!
Functional models, e.g. C/C++ programs

Could be synthesized using a High-Level Synthesis tool?
Use Model: SystemC as Glue!

- Transaction-level modeling is communication-centric
Typical Use Case: Virtual Platform

Multiple software stacks

CPU
ROM
RAM
DMA

Interrupt
Timer
I/O
Bridge

DSP
ROM
RAM

Interrupt
Timer
A/D

Bridge

I/O
Memory interface
RAM
DMA

Custom peripheral
D/A

Multiple buses and bridges

Digital and analog hardware IP blocks

© Accellera Systems Initiative
## Virtual Platform Characteristics

<table>
<thead>
<tr>
<th>Instruction Set Simulator or software stubs</th>
<th>Transaction-Level Model</th>
<th>RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available early</td>
<td>Available early</td>
<td>✓</td>
</tr>
<tr>
<td>Fast enough to run applications</td>
<td>Fast enough to run applications</td>
<td>✓</td>
</tr>
<tr>
<td>Little or no hardware detail</td>
<td>Register-accurate</td>
<td>✓</td>
</tr>
<tr>
<td>No timing information</td>
<td>Some timing information</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Notes:
- ✓: Available
- ❌: Not Available
What is TLM Used For?

Accelerates product release schedule

Firmware / software

Fast enough

TLM

Ready before RTL

RTL

Test bench

Software development

Architectural exploration

Hardware (co)verification

TLM = golden model
Virtual Prototyping using SystemC and TLM-2.0

• Introduction
• Development of TLM-2.0
• TLM-2.0 Sockets and Interfaces
• LT, AT, and CA
• Generic Payload and Extensions
• Interoperability
TLM-2 Requirements

• Focus on memory-mapped bus modeling
• Not meant to exclude non-bus protocols

Speed!

Interoperability!
Multiple Abstraction Levels

- **FUNCTIONAL VIEW**
  - Algorithm developer
  - Untimed

- **ARCHITECTURE VIEW**
  - Tuning the platform
  - Approximately-timed

- **VERIFICATION VIEW**
  - Functional verification
  - Untimed through Cycle Accurate

- **PROGRAMMERS VIEW**
  - Software developer
  - Loosely-timed

- **RTL**
  - Implementation

- Can mix-and-match
Use Cases, Coding Styles and Mechanisms

Use cases
- Software development
- Software performance
- Architectural analysis
- Hardware verification

TLM-2 Coding styles (just guidelines)
- Loosely-timed
- Approximately-timed

Mechanisms (definitive API for TLM-2.0)
- Blocking interface
- DMI
- Quantum
- Sockets
- Generic payload
- Phases
- Non-blocking interface
Coding Styles

• **Loosely-timed** = as fast as possible
  - Only sufficient timing detail to boot O/S and run multi-core systems
  - Processes can run ahead of simulation time (temporal decoupling)
  - Each transaction completes in one function call
  - Uses direct memory interface (DMI)

• **Approximately-timed** = just accurate enough for performance modeling
  - *aka* cycle-approximate or cycle-count-accurate
  - Sufficient for architectural exploration
  - Processes run in lock-step with simulation time
  - Each transaction has 4 timing points (extensible)

• Guidelines only – not definitive
The TLM 2.0 Classes

Interoperability layer for bus modeling

- Generic payload
- Phases
- Initiator and target sockets

TLM-1 standard

TLM-2 core interfaces:
- Blocking transport interface
- Non-blocking transport interface
- Direct memory interface

IEEE 1666™ SystemC

Utilities:
- Convenience sockets
- Payload event queues
- Quantum keeper

Analysis ports

Analysis interface
Interoperability Layer

1. Core interfaces and sockets

Initiator → Target

2. Generic payload

- Command
- Address
- Data
- Byte enables
- Response status
- Extensions

3. Base protocol

- BEGIN_REQ
- END_REQ
- BEGIN_RESP
- END_RESP

- Maximal interoperability for memory-mapped bus models

Either write bytes

or read bytes
Utilities

Interoperability layer

Initiator

- Core interfaces
- Sockets
- Generic payload
- Base protocol

Target

Coding Style
Loosely- or Approximately-timed

Utilities

- Convenience sockets
- Quantum keeper (LT)
- Payload event queues (AT)
- Instance-specific extensions (GP)

- Productivity
- Shortened learning curve
- Consistent coding style
Virtual Prototyping using SystemC and TLM-2.0

• Introduction
• Development of TLM-2.0
• TLM-2.0 Sockets and Interfaces
• LT, AT, and CA
• Generic Payload and Extensions
• Interoperability
Initiators and Targets

References to a single transaction object are passed along the forward and backward paths.
TLM-2 Connectivity

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

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

Sockets provide fw and bw paths, and group interfaces

**Initiator socket**

- *class tlm-fw_transport_if* :
  - b_transport() 
  - nb_transport_fw() 
  - get_direct_mem_ptr() 
  - transport_dbg()

**Target socket**

- *class tlm-bw_transport_if* :
  - nb_transport_bw() 
  - invalidate_direct_mem_ptr()
Sockets, Ports and Exports

Initiator or Interconnect

Port

Export

Forward path
Return path
Backward path
Return path

Export

Port

Target or Interconnect

Target socket
Socket Binding and Interfaces

- **Backward interface**
- **Forward interface**

**Initiator socket**
- Initiator or Interconnect
- Port
- Export
  - Interface method calls
- Forward path
  - Return path
- Backward path
  - Forward path

**Export**
- Target or Interconnect
- Port
- Export
  - Interface method calls
- Forward path
  - Return path
- Backward path
  - Forward path

bind

© Accellera Systems Initiative 2014
Virtual Prototyping using SystemC and TLM-2.0

- Introduction
- Development of TLM-2.0
- TLM-2.0 Sockets and Interfaces
- LT, AT, and CA
- Generic Payload and Extensions
- Interoperability
Software Execution and Simulation

Software source code → Binary executable
- Interpretation
  - ISS on Host
  - Simulator
- Native execution
  - Physical Target CPU
  - Physical System
- Dynamic translation / JIT
  - Fast ISS on Host
  - Simulator
- Cross-compiled → Binary executable
  - Native execution
  - Host computer
  - Simulator
Software Execution without Sync

Executing software

Initiator model does not yield

Other initiators do not get to run

Simulated resources:
Bus
Memory
Peripherals

CPU ISS Initiator

CPU ISS Initiator

CPU ISS Initiator
Software Execution with Sync

Explicit synchronization between software threads

Synchronization is independent of platform timing
Software Execution with a Quantum

Executing software

CPU ISS Initiator

CPU ISS Initiator

CPU ISS Initiator

Simulated resources:
- Bus
- Memory
- Peripherals

Initiators use a time quantum

Each initiator gets a time slice

Resources consume simulation time

© Accellera Systems Initiative
Causality with `b_transport`

- **Initiator** sets attributes
- **Interconnect** modifies address
- **Target** modifies attributes
- **Initiator** checks response
- **Interconnect** returns
- **Target** returns
Loosely-Timed Components

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

- Inter-process communication is annotated with delays
- Each process is synchronized using the SystemC scheduler
- Delays can be accurate or approximate
AT and CA

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

**AT**

- BEGIN_REQ
- END_REQ
- BEGIN_RESP
- END_RESP

Wake up at significant timing points

**CA**

Wake up every cycle
Virtual Prototyping using SystemC and TLM-2.0

• Introduction
• Development of TLM-2.0
• TLM-2.0 Sockets and Interfaces
• LT, AT, and CA
• Generic Payload and Extensions
• Interoperability
The Generic Payload

• Typical attributes of memory-mapped busses
  • reads, writes, byte enables, single word transfers, burst transfers, streaming

• Off-the-shelf general purpose payload
  • for abstract bus modeling
  • * ignorable* extensions allow full interoperability

• Used to model specific bus protocols
  • mandatory static extensions
  • compile-time type checking to avoid incompatibility
  • low implementation cost when bridging protocols

Specific protocols can use the same generic payload machinery
# Generic Payload Attributes

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

- There are defaults, but transaction objects are typically pooled
- Initiator must set all attributes except byte enable length and extensions

Array owned by initiator

Array owned by initiator

Ignored if ptr = 0

Must be > 0

Try DMI!
The Extension Mechanism

- Every generic payload has an array-of-extension-pointers
- One pointer per extension type, initialized by the constructor
Using a Memory Manager

Allocate transaction object

3

acquire()

nb_transport_fw

Return

nb_transport_bw

Call

acquire()

nb_transport_bw

Call

release()

nb_transport_bw

Return

nb_transport_bw

Return

acquire()

nb_transport_bw

Return

release()

free()
Example Topology

- Memory manager
- Initiator
- Initiator
- Initiator
- Thread
- Thread
- Bus Interconnect
- Controller
- Target
- Target
- Target
- Target
- Target
- Target
- Target
- Target
- Target
- Multi-sockets

Data
Virtual Prototyping using SystemC and TLM-2.0

- Introduction
- Development of TLM-2.0
- TLM-2.0 Sockets and Interfaces
- LT, AT, and CA
- Generic Payload and Extensions
- Interoperability
First Kind of Interoperability

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

```c
// ... snip ...

// TLM type definitions

// Tlm_initiator_socket<32, Tlm_base_protocol_types>
my_socket;
```

- Functional incompatibilities are still possible (e.g. writing to a ROM)
Second Kind of Interoperability

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

```cpp
tlm_initiator_socket<32, my_protocol> my_socket;
```

- 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
Bridges

tlm_*_socket< 32, protocol_A >

Initiator

Bridge

depth_copy_from()

update_original_from()

Target

tlm_*_socket< 32, protocol_B >

Generic Payload

Extension

Extension

Generic Payload

Extension

Extension

Generic Payload

Extension

Extension

Generic Payload

Extension

Extension

Could pass the same transaction if their lifetimes allow
Levels of Use

1. Buy models with the TLM-2.0 sticker
2. Write LT components
   **Beware: requires more expertise...**
3. Write AT components
4. Support LT/AT switching
Missing from TLM-2.0?

- Interrupts
- Other wire-level interfaces
- Register address maps
- Parameterization of TLM IP
- TL power models
Associated Standards and Activities

• Accellera Configuration, Control and Inspection (CCI) Working Group

• Synopsys Virtualizer & SCML, Mentor Vista, Cadence VSP
  ARM FastModels, Carbon, Sonics, Arteris, OVP, ...

• TLM Central  www.dr-embedded.com/tlmcentral
Questions?

For Further Information

http://www.doulos.com/knowhow/systemc/tlm2