Pushing the Limits of Standard-Compliant Parallel SystemC Simulation

Rainer Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine
A non-exclusive, irrevocable, royalty-free copyright permission is granted by Rainer Doemer, CECS, to use this material in developing all future revisions and editions of the resulting draft and approved Accellera Systems Initiative SystemC standard, and in derivative works based on this standard.
Outline

4. Pushing the Limits of
3. Standard-Compliant
2. Parallel
1. SystemC Simulation
1. SystemC Simulation

• Discrete Event Simulation (DES)
  – Concurrent threads of execution
  – Managed by a central scheduler
  – Driven by events and time advances
    • Delta-cycle
    • Time-cycle
  ➢ Partial temporal order with barriers

• Example
  – Accellera Proof-of-Concept Simulator
  ➢ Sequential, slow!
2. Parallel SystemC Simulation

- **Parallel** Discrete Event Simulation (PDES)
  - Concurrent threads of execution
  - Managed by a central scheduler
  - Driven by events and time advances
    - Delta-cycle
    - Time-cycle
  ➢ Synchronous parallelism
  ➢ Threads execute in parallel *iff*
    - in the same delta cycle, *and*
    - in the same time cycle
  ➢ *Order of magnitude faster!*
3. Standard-Compliant Parallel SystemC Simulation

- IEEE Standard 1666™-2011
  - Revision of IEEE Std. 1666-2005

... unfortunately stands in the way of parallel SystemC simulation!

- SystemC Evolution Day 2016
  - “Seven Obstacles in the Way of Parallel SystemC Simulation”, Rainer Doemer, Munich, Germany, May 2016.

- SystemC standard
  - ... must embrace true parallelism
  - ... must evolve in a major revision (3.x)
4. Pushing the Limits …

• While the SystemC standard has not changed, my group has worked hard
  ➢ “Let’s make the best of it!”

• Goals
  – Accept SystemC as it is (well, most of it)
  – Build the best parallel SystemC simulator possible
  – Aim for maximum compliance with the standard

➢ We took this risk, and created RISC!
  ➢ Recoding Infrastructure for SystemC

➢ RISC pushes the limits
to overcome the 7 obstacles …
Obstacle 1: Co-Routine Semantics

• Fact: IEEE 1666-2011 requires *co-operative multitasking*

  ➢ Quotes from Section “4.2.1.2 Evaluation phase” (pages 17, 18):

  Since process instances execute without interruption, *only a single process instance can be running at any one time*, [...]. A process shall not pre-empt or interrupt the execution of another process. This is known as *co-routine semantics* or *co-operative multitasking*. [...]

  The scheduler is *not pre-emptive*. An application can assume that a method process will execute in its entirety without interruption, and a thread or clocked thread process will execute the code between two consecutive calls to function *wait* without interruption.

• Problem: Uninterrupted execution guarantee

  An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the *co-routine semantics* defined in this subclause. In other words, the implementation would be obliged to analyze any dependencies between processes and to constrain their execution to match the co-routine semantics.

• Proposal: Explicitly allow parallel execution, preemption
  – Process instances at the same time \((t,\delta)\) may execute in parallel
    • Model designer must write thread safe code, avoid race conditions
      ➢ Parallel systems, parallel models, parallel programming

Seven Obstacles in the Way of Parallel SystemC Simulation  (c) 2016 R. Doemer, CECS
Pushing the Limits with RISC

- Obstacle 1: *Resolved!*
  - Introduce a dedicated SystemC Compiler
    - Automatic analysis of parallel access conflicts
    - Run SystemC processes in parallel if there are no conflicts
    - Faster simulation
    - Results remain the same

---

**Obstacle 1: Co-Routine Semantics**

- Fact: IEEE 1666-2011 requires co-operative multitasking
  - Quotes from Section “4.2.1.2 Evaluation phase” (pages 17, 18):
    Since process instances execute without interruption, *only a single process instance can be running at any one time.* [...]. A process shall not pre-empt or interrupt the execution of another process. This is known as co-routine semantics or co-operative multitasking.
    [...] The scheduler is not pre-emptive. An application can assume that a method process will execute in its entirety without interruption, and a thread or clocked thread process will execute the code between two consecutive calls to function wait without interruption.

- Problem: Uninterrupted execution guarantee
  - An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the co-routine semantics defined in this subclause. In other words, the implementation would be obliged to analyze any dependencies between processes and to constrain their execution to match the co-routine semantics.

- Proposal: Explicitly allow parallel execution, preemption
  - Process instances at the same time (t,δ) may execute in parallel
    - Model designer must write thread safe code, avoid race conditions
    - Parallel systems, parallel models, parallel programming
Obstacle 2: Simulator State

• Fact: Discrete Event Simulation (DES) is presumed
  ➢ Example from IEEE 1666-2011, page 31: \texttt{sysc/kernel/sc_simcontext.h}

  
  \begin{verbatim}
  ...]
  bool sc_pending_activity_at_current_time();
  bool sc_pending_activity_at_future_time();
  bool sc_pending_activity();
  bool sc_time_to_pending_activity();
  [...]
  \end{verbatim}

• Problem: Parallel Discrete Event Simulation (PDES) is different from sequential DES
  – After elaboration, there may be \textit{multiple running threads}
  – Scheduling may happen while some threads are still running

• Proposal: Carefully review simulator state primitives and revise as needed for PDES
  ➢ Adapt the functions and APIs for parallel execution semantics
  ➢ The general notion of \textit{shared state} needs attention…
Pushing the Limits with RISC

• Obstacle 2: Ongoing…
  ➢ Review and revise the SystemC API
    ➢ Slightly adjust the semantics
    ➢ Maximize compliance with standard
  ➢ For APIs on the slide:
    ➢ User’s expectations can be met
    ➢ Example: SystemC Integration with Simics VP works fine
Obstacle 3: Lack of Thread Safety

• Fact: Primitives are generally not multi-thread safe
  ➢ Suspicious example from IEEE 1666-2011, page 194:

  ```
  [...]
  sc_length_param   length10(10);
  sc_length_context cntxt10(length10); // length10 now in context
  sc_int_base       int_array[2];      // Array of 10-bit integers
  [...]
  ```

• Problem: Parallel execution may lead to race conditions
  – Race conditions result in non-deterministic/undefined behavior
  – Explicit protection (e.g. by mutex locks) is cumbersome
  – Identifying problematic constructs is difficult
    • Example: `class sc_context`, commented as “co-routine safe”

• Proposal: Require all primitives to be multi-thread safe
  – Carefully revise the proof-of-concept SystemC library
  ➢ Encouraging item: `async_request_update` is thread-safe!
    ➢ See “5.15 sc_prim_channel”, IEEE 1666-2011, page 121
Pushing the Limits with RISC

- Obstacle 3: Ongoing...
  - Revise SystemC primitives for multi-thread safety
    - Protection by inserted locks
    - Store state in local or thread-local storage
    - For deterministic debugging, user can control number of parallel threads (e.g. set to 1)

Obstacle 3: Lack of Thread Safety

- Fact: Primitives are generally not multi-thread safe
  - Suspicious example from IEEE 1666-2011, page 194:
    ```
    [...] 
    sc_length_param length10(10);
    sc_length_context cntxt10(length10); // length10 now in context 
    sc_int_base int_array[2]; // Array of 10-bit integers 
    [...] 
    ```

- Problem: Parallel execution may lead to race conditions
  - Race conditions result in non-deterministic/undefined behavior
  - Explicit protection (e.g. by mutex locks) is cumbersome
  - Identifying problematic constructs is difficult
    - Example: `class sc_context, commented as "co-routine safe"

- Proposal: Require all primitives to be multi-thread safe
  - Carefully revise the proof-of-concept SystemC library
  - Encouraging item: `async_request_update` is thread-safe!
    - See “5.15 sc_prim_channel”, IEEE 1666-2011, page 121

(c) 2019 R. Doemer, CECS
Obstacle 4: Class sc_channel

• Fact: sc_channel is an alias type for sc_module
  ➢ IEEE 1666-2011, Section “5.2.23 sc_behavior and sc_channel” (page 56):
    
    The typedefs sc_behavior and sc_channel are provided for users to express their intent. 
    NOTE—There is no distinction between a behavior and a hierarchical channel 
    other than a difference of intent. Either may include both ports and public member functions.

  ➢ systemc-2.3.1/include/sysc/kernel/sc_module.h

    [...] 
    typedef sc_module sc_channel; 
    typedef sc_module sc_behavior; 
    [...] 

• Problem: Alias type is only another name, no new type
  ➢ Language does not distinguish modules and channels
  ➢ No separation of communication and computation
    ➢ Breaks a key system-level design principle…

• Proposal: Class sc_channel, derived from sc_module
  ➢ Module encapsulates computation (hosts threads/processes)
  ➢ Channel encapsulates communication (implemented interfaces)
Pushing the Limits with RISC

- Obstacle 4: Fixed!
  - Derive `sc_channel` from base class `sc_module`
    - Minimal change in SystemC headers
    - Two different types at compile-time
    - Easy distinction in static analysis
    - No known negative side-effects

Obstacle 4: Class `sc_channel`

- Fact: `sc_channel` is an alias type for `sc_module`
  - IEEE 1666-2011, Section "5.2.23 sc_behavior and sc_channel" (page 56):
    ```
    The typedefs sc_behavior and sc_channel are provided for users to express their intent.
    NOTE—There is no distinction between a behavior and a hierarchical channel
    other than a difference of intent. Either may include both ports and public member functions.
    ```
  - `systemc-2.3.1/include/sysc/kernel/sc_module.h`
    ```
    [...]
    typedef sc_module sc_channel;
    typedef sc_module sc_behavior;
    [...]```
- Problem: Alias type is only another name, no new type
  - Language does not distinguish modules and channels
  - No separation of communication and computation
    - Breaks a key system-level design principle...
- Proposal: Class `sc_channel`, derived from `sc_module`
  - Module encapsulates computation (hosts threads/processes)
  - Channel encapsulates communication (implemented interfaces)
Obstacle 5: TLM-2.0

• Fact: Channel concept has disappeared

• Problem:
  Where is the channel?
  – Interface methods are well-defined, but not contained
  – Separation of concerns “Computation ≠ Communication” principle is broken
  – Proposal:
    Encapsulate communication methods in channels
Pushing the Limits with RISC

• Obstacle 5: *Reevaluated, Resolved!*
  - Socket Call Path (SCP) analysis
  - Variable Entanglement analysis
  - Compile-time analysis can identify target methods executed by TLM-2.0 calls
  - Support for interconnect modules and DMI

---

Obstacle 5: TLM-2.0

• Fact: Channel concept has disappeared
  - "The Definitive Guide to SystemC: TLM-2.0 and the IEEE 1666-2011 Standard"; Presentation by David Black, Doulas, at DAC’15 Training Day

• Problem:
  Where is the channel?
  - Interface methods are well-defined, but not contained
  - Separation of concerns "Computation ≠ Communication" principle is broken
  - Proposal:
    Encapsulate communication methods in channels

---

SystemC Evolution Day, Oct. 31, 2019

(c) 2019 R. Doemer, CECS
Obstacle 6: Sequential Mindset

- Fact: SC_METHOD is preferred over SC_THREAD, context switches are considered overhead
  - IEEE 1666-2011, Section 5.2.11 on threads (page 44):
    - Each thread or clocked thread process requires its own execution stack.
    - As a result, context switching between thread processes may impose a simulation overhead when compared with method processes.

- Problem: Sequential modeling is encouraged
  - However, systems are parallel by nature, so should be models
  - Avoiding context switches is the wrong optimization criterion

- Proposal: Use actual threads, eliminate SC_METHOD, identify dependencies among threads
  - Promote parallel mindset, with true thread-level parallelism
    - Speed due to parallel execution, not due to fewer context switches
  - Explicitly express task relations (use `e.notify()`, `wait(e)`)
    - Synchronize, communicate through events and channels
Pushing the Limits with RISC

• Obstacle 6: *Not a problem*
  ➢ **SC_METHOD**, **SC_THREAD**, **SC_CTHREAD** can all be supported
  ➢ Static analysis per process type
  ➢ **SC_METHOD** execution by dedicated invoker threads
  ➢ Nice optimization problem for efficient grouping with minimal conflicts

---

**Obstacle 6: Sequential Mindset**

• Fact: **SC_METHOD** is preferred over **SC_THREAD**, context switches are considered overhead
  ➢ IEEE 1666-2011, Section 5.2.11 on threads (page 44):
    
    Each thread or clocked thread process requires its own execution stack.
    As a result, context switching between thread processes may impose a simulation overhead when compared with method processes.

• Problem: Sequential modeling is encouraged
  – However, systems are parallel by nature, so should be models
  – Avoiding context switches is the wrong optimization criterion

• Proposal: Use actual threads, eliminate **SC_METHOD**, identify dependencies among threads
  ➢ Promote parallel mindset, with true thread-level parallelism
    • Speed due to parallel execution, not due to fewer context switches
  ➢ Explicitly express task relations (use `e.notify()`, `wait(e)`)
    • Synchronize, communicate through events and channels
Obstacle 7: Temporal Decoupling

- Fact: TD is designed to speed up sequential DES
  - IEEE 1666-2011, Section 12.1 on “TLM-2.0 global quantum” (page 453):
    - Temporal decoupling permits SystemC processes to run ahead of simulation time for an amount of time known as the time quantum and is associated with the loosely-timed coding style. Temporal decoupling permits a significant simulation speed improvement by reducing the number of context switches and events.
    - Abstraction trades off accuracy for higher simulation speed

- Problem: PDES is a different foundation than DES
  - TD design assumptions are not necessarily true for PDES
  - Global time quantum is a technical obstacle (race condition)

- Proposal: Reevaluate costs/benefits, redesign if needed
  - Analyze TD idea for PDES, adopt advantages, drop drawbacks
    - Avoid tlm_global_quantum, promote wait(time)
  - Consider the use of a compiler to optimize scheduling, timing
    - Out-of-Order PDES is one solution (fully automatic, accurate)
Pushing the Limits with RISC

Obstacle 7: TBD…

➢ Investigate in future work

➢ Is there any need for this abstraction in PDES?

➢ Out-of-order PDES

➢ Likely can achieve the same benefit

➢ Without loss of accuracy

➢ Global time quantum (if needed) can be protected by mutex

Obstacle 7: Temporal Decoupling

➢ Fact: TD is designed to speed up sequential DES
  ➢ IEEE 1666-2011, Section 12.1 on “TLM-2.0 global quantum” (page 453):
    - Temporal decoupling permits SystemC processes to run ahead of simulation time for an amount of time known as the time quantum and is associated with the loosely-timed coding style. Temporal decoupling permits a significant simulation speed improvement by reducing the number of context switches and events.

➢ Problem: PDES is a different foundation than DES
  - TD design assumptions are not necessarily true for PDES
  - Global time quantum is a technical obstacle (race condition)

➢ Proposal: Reevaluate costs/benefits, redesign if needed
  - Analyze TD idea for PDES, adopt advantages, drop drawbacks
    - Avoid tlm_global_quantum, promote wait(time)
  - Consider the use of a compiler to optimize scheduling, timing
    - Out-of-Order PDES is one solution (fully automatic, accurate)
• **Out-of-Order** Parallel DES
  – Threads execute in parallel *iff*
    • in the same delta cycle, *and*
    • in the same time cycle,
    • *OR* if there are no conflicts!
  ➢ Breaks synchronization barrier!
  ➢ Threads run as soon as possible, even ahead of time
  ➢ Maximum speedup!
    • Results at [DATE’12], [IEEE TCAD’14]
  ➢ Our approach preserves…
    ➢ Cause and effect relationship
    ➢ Accuracy in results and timing
    ➢ Maximum compliance with standard
Recoding Infrastructure for SystemC

- **RISC Infrastructure**
  - Dedicated RISC compiler tool chain
  - Compliance with standard SystemC semantics
  - Open source available from CECS

- **Out-of-order Parallel Simulation**
  - Fully accurate
  - Two orders of magnitude faster

---

Tradional SystemC Simulation

- Accelera Library
- SystemC Model
- Header File
- C++ Compiler
- Executable
- Host PC
  - Sequential (1x)

RISC OoO Parallel Simulation

- RISC Library
- SystemC Model
- Header File
- RISC Compiler
- Parallel Executable
- Multi- / Many-Core Host
  - Out-of-order Parallel (10x – 100x)

**212x speedup [DAC’17]**
Scaling RISC: File Hierarchies, 3rd Party IP

Traditional SystemC Simulation
- `system.c.h` Header File
- Accellera Library
- `system.c.h` SystemC Model
- C++ Compiler
- Executable
- Host PC
- Sequential (1x)

RISC OoO Parallel Simulation
- `system.c.h` Header File
- RISC Library
- RISC Model
- RISC Compiler
- Parallel Executable
- Multi- / Many-Core Host
- Out-of-order Parallel (10x – 100x)

Scalable RISC OoO Parallel Simulation
- `system.c.h` Header File
- RISC Library
- RISC Model
- SystemC Top
- RISC Compiler
- Combined Parallel Executable
- Multi- / Many-Core Host
- Out-of-order Parallel (10x – 100x)
- Component Libraries
- 3rd Party IP
- IP Integration and Protection
- Frontend Tools, CoFluent™ Studio

Frontend Tools, Integration
- New Support for Partial Segment Graphs (PSG)

Component Libraries
- 3rd Party Library

(c) 2019 R. Doemer, CECS

SystemC Evolution Day, Oct. 31, 2019
Scaling RISC: Support for TLM-2.0

- Various Modeling Styles Supported by RISC v0.6.0

Structural Composition  Synchronization  Connectivity

Explicit Memories  Interconnect Modules  DMI

Video Codec  Left Audio Codec  Right Audio Codec

Video Monitor  Left Speaker  Right Speaker

Video Codec  Left Audio Codec  Right Audio Codec

Video Monitor  Left Speaker  Right Speaker

Video Codec  Left Audio Codec  Right Audio Codec

Video Monitor  Left Speaker  Right Speaker

Video Codec  Left Audio Codec  Right Audio Codec

Video Monitor  Left Speaker  Right Speaker

Video Codec  Left Audio Codec  Right Audio Codec

Video Monitor  Left Speaker  Right Speaker

(c) 2019 R. Doemer, CECS
Scaling RISC: Analysis and Transformation

- Example: Model Visualization
  - Hierarchy and connectivity
    - Ports and sockets
  - Threads in modules
RISC Open Source

• RISC Compiler and Simulator, Release V0.6.0
  ➢ http://www.cecs.uci.edu/~doemer/risc.html#RISC060
    • Installation notes and script: INSTALL, Makefile
    • Open source tar ball: risc_v0.6.0.tar.gz
    • Docker script and container: Dockerfile
    • Doxygen documentation: RISC API, OOPSC API
    • Tool manual pages: risc, simd, visual, ...
    • BSD license terms: LICENSE

  – Companion Technical Report

➢ Docker container:
  ➢ https://hub.docker.com/r/ucirvinelecs/risc060/
Conclusion

• Overcoming Obstacles towards Parallel SystemC
  1. Co-Routine Semantics:  *Resolved*
  2. Simulator State:  *Ongoing…*
  3. Lack of Thread Safety:  *Ongoing…*
  4. Class sc_channel:  *Fixed*
  5. TLM-2.0:  *Reevaluated, Resolved*
  6. Sequential Mindset:  *Not a problem*
  7. Temporal Decoupling:  *TBD…*

• Recoding Infrastructure for SystemC
  – Introduction of a dedicated SystemC compiler
  – Out-of-order parallel simulation on multi- and many-core hosts
  – Maximum compliance with IEEE SystemC semantics

• Open Source
  – Thanks to Intel Corporation!
References (1)

  ➢ Reprint to appear as journal article in ACM Transactions on Embedded Computer Systems!
References (2)

References (3)

Acknowledgments

• For solid work, fruitful discussions, and honest feedback, I would like to thank:
  – My team at UCI
    • Zhongqi Cheng, Daniel Mendoza, Emad Arasteh
    • Tim Schmidt, Guantao Liu
    • Farah Arabi, Spencer Kam
  – Our collaborators at Intel
    • Ajit Dingankar
    • Desmond Kirkpatrick
    • Abhijit Davare
    • Philipp Hartmann
  – And many others…
• This work has been supported in part by substantial funding from Intel Corporation. Thank you!