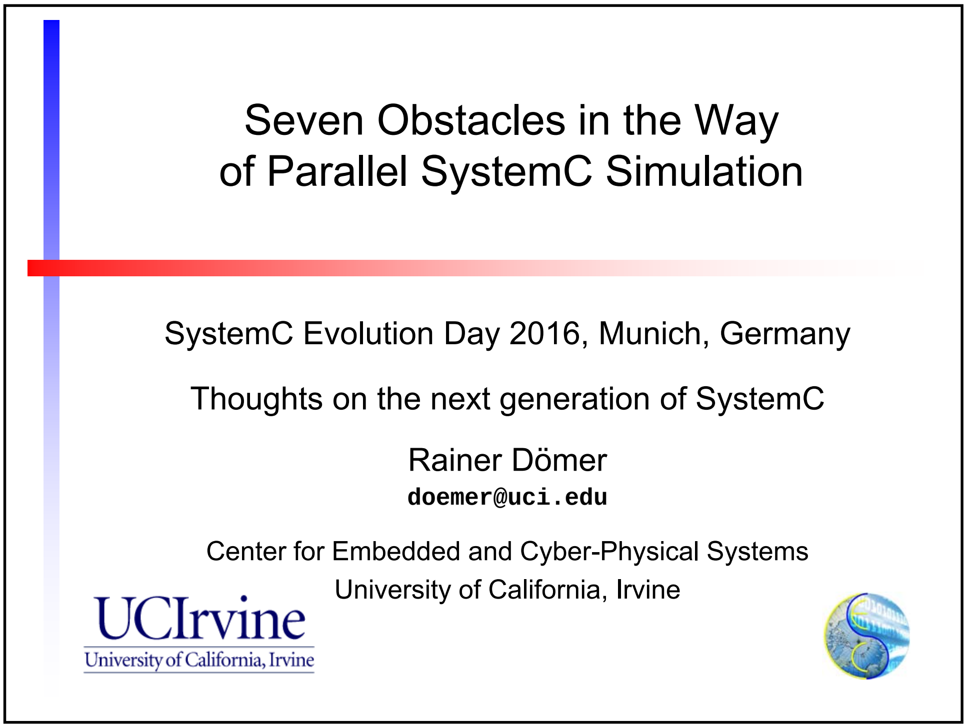


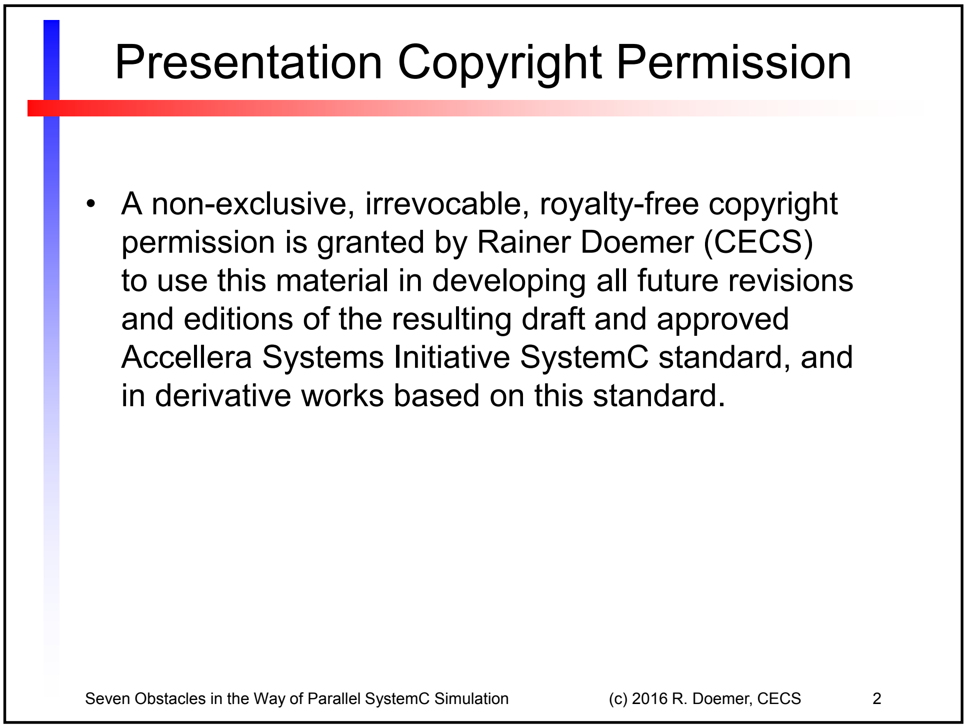
Seven Obstacles in the Way of Parallel SystemC Simulation

SystemC Evolution Day 2016, Munich, Germany

Thoughts on the next generation of SystemC

Rainer Dömer
`doemer@uci.edu`

Center for Embedded and Cyber-Physical Systems
University of California, Irvine



Presentation Copyright Permission

- 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.

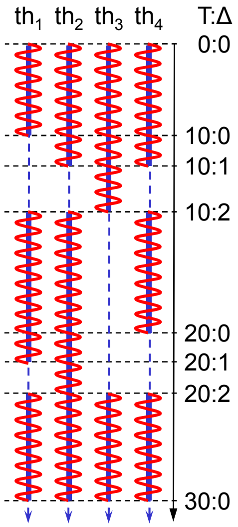
Goals

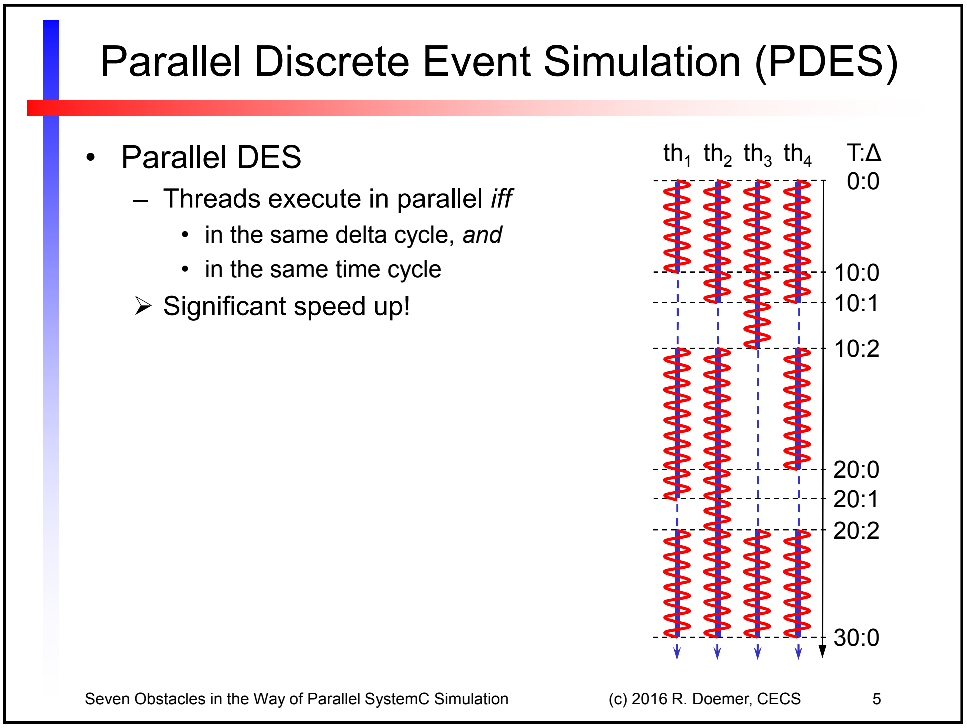
- Truly parallel simulation of SystemC models
  - – High speed due to parallel execution on multi/many core hosts
  - – Compliant to the IEEE 1666 standard
- Identify the main obstacles in the way of standard-compliant parallel SystemC
  - – And propose potential solutions
- Technical review and evaluation of
  - – Standard SystemC® Language Reference Manual
    - IEEE Std 1666™-2011 (Revision of IEEE Std 1666-2005)
  - – Accellera open source proof-of-concept library (v2.3.1)

➢ Warning: Controversial Content Ahead!
  - – Evolve SystemC to true parallelism (major revision)
  - – Let's have a good discussion!

Discrete Event Simulation (DES)

- Traditional 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
- Sequential Reference Simulator
  - – SystemC standard IEEE 1666-2011
  - ➢ A single thread is active at any time
  - ➢ Cannot exploit parallelism
  - ➢ Cannot utilize multiple cores





Parallel Discrete Event Simulation (PDES)

- Parallel DES
  - – Threads execute in parallel *iff*
    - in the same delta cycle, *and*
    - in the same time cycle
  - ➢ Significant speed up!

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.

Parallel Discrete Event Simulation (PDES)

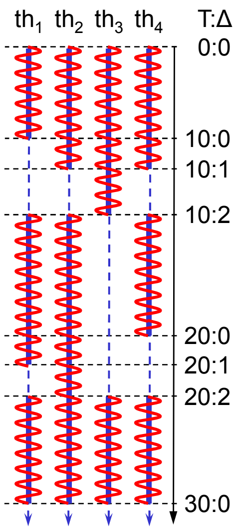
- Parallel DES
  - – Threads execute in parallel *iff*
    - in the same delta cycle, *and*
    - in the same time cycle
  - ➢ Significant speed up!

➢ SystemC LRM Requirement:
“The scheduler is not pre-emptive.”

```
int x;  // global variable

void thread1()       void thread42()
{ x = 0;             { x = 7;
  x = x + 1;           x = x * 6;
  cout << x;           cout << x;
}                    }
```

- ➢ SystemC: guaranteed safe!
- ➢ PDES: not safe! (race condition)



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: `sysc/kernel/sc_simcontext.h`

```
[...]
bool sc_pending_activity_at_current_time();
bool sc_pending_activity_at_future_time();
bool sc_pending_activity();
bool sc_time_to_pending_activity();
[...]
```

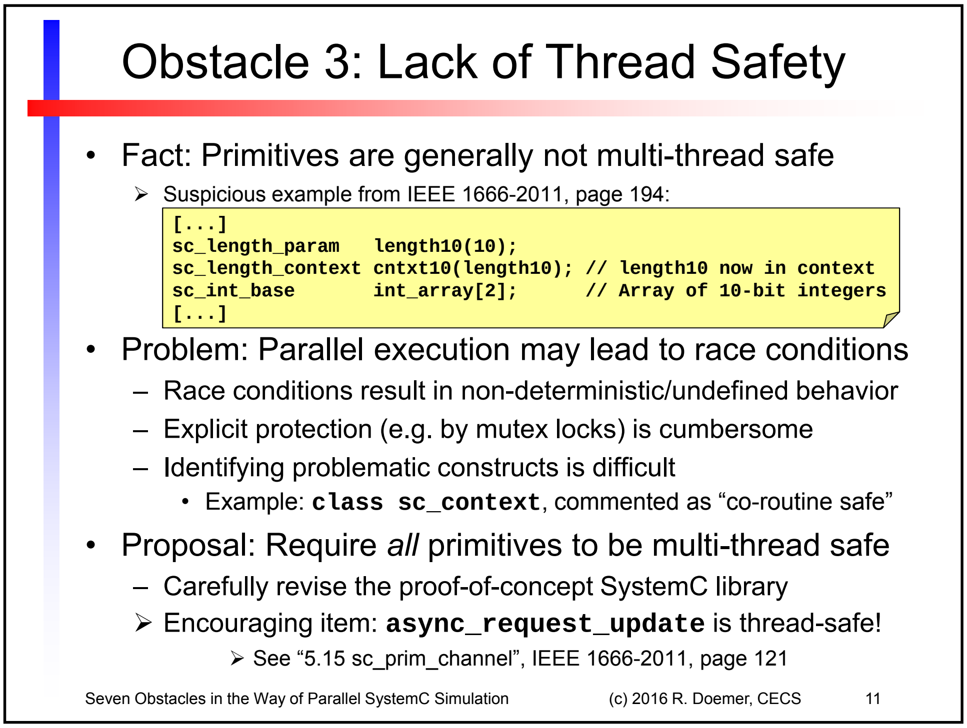
- Problem: Parallel Discrete Event Simulation (PDES) is different from sequential DES
  - – After elaboration, there may be *multiple running threads*
  - – Scheduling may happen while some threads are still running
- Proposal: Carefully review simulator state primitives and revise as needed for PDES
  - ➢ Adapt the functions and APIs for parallel execution semantics
  - ➢ The general notion of *shared state* needs attention…

Obstacle 2: Simulator State

- Fact: Discrete Event Simulation (DES) is presumed
- Problem: Parallel Discrete Event Simulation (PDES) is different from sequential DES
- Proposal: Carefully review simulator state primitives and revise as needed for PDES
  - ➢ The general notion of *shared state* needs attention
  - ➢ Special consideration for very strict semantics, e.g. debugging: Quote from IEEE 1666-2011, Section "4.2.1.2 Evaluation phase" (page 17):

> The order in which process instances are selected from the set of runnable processes is implementation defined. However, if a specific version of a specific implementation runs a specific application using a specific input data set, the order of process execution shall not vary from run to run.

  - ➢ Strict DES can remain valid as a special case of PDES
    - ➢ While PDES typically runs up to n threads in parallel, where n = number of cores on the host, we can set n = 1 to mimic the classic DES case

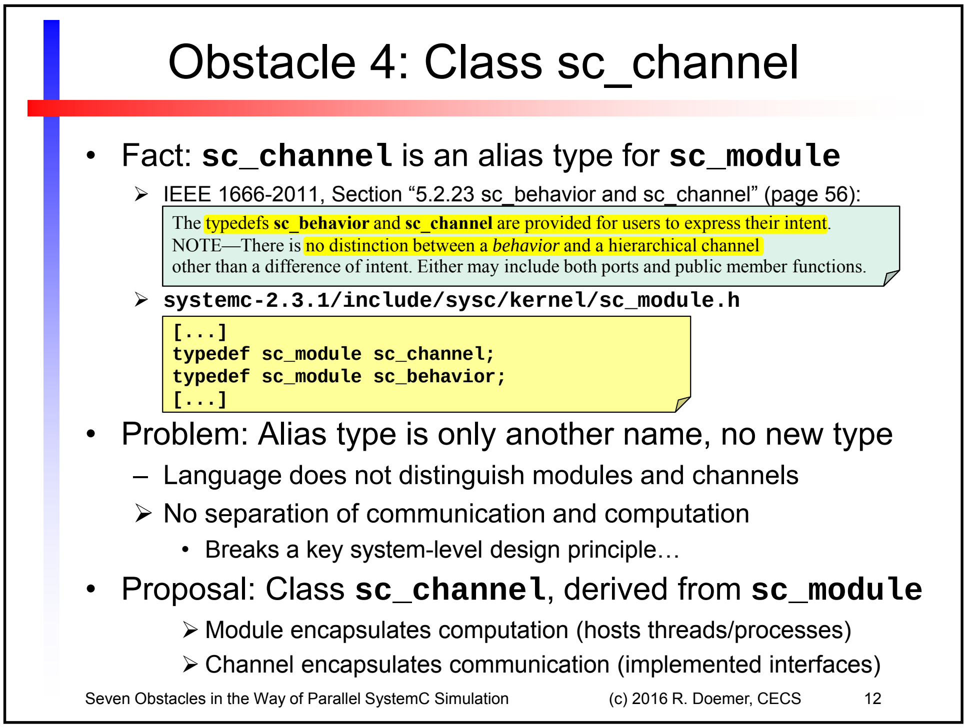


Obstacle 3: Lack of Thread Safety

- 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



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 4: Class sc_channel

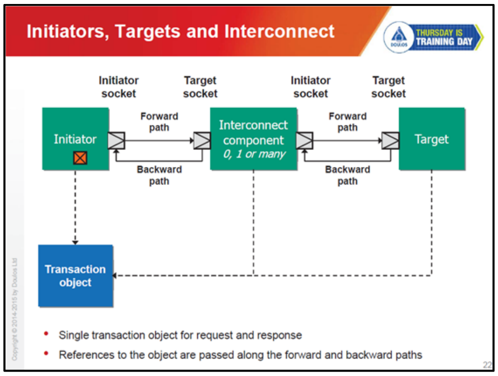
- Proposal: Class **`sc_channel`**, derived from **`sc_module`**
  - ➢ Module encapsulates computation (hosts threads/processes)
  - ➢ Channel encapsulates communication (implemented interfaces)
- ➢ Q: Why do we need channels? A: Thread safe communication!
- – Example: Blocking write in primitive channel **`sc_fifo.h`**

```
template <class T> inline
void sc_fifo<T>::write( const T& val_ )
{   sc_stacked_lock l(m_mutex);  // lock the channel mutex
    while( num_free() == 0 ) {
        sc_core::wait( m_data_read_event );
    }
    m_num_written ++;
    buf_write( val_ );
    request_update();
}
```

  - ➢ Race condition between **`num_free`** and **`m_num_written`**
  - ➢ Prevented by locking **`m_mutex`** of this channel instance
- – Channel acts as a *monitor* for multi-thread safe communication

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, Doulos, at DAC'15 Training Day
- Problem: Where is the channel?

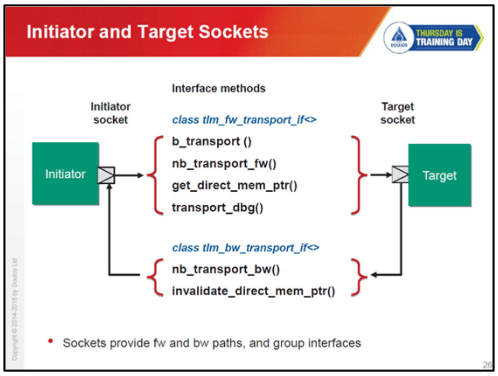


Initiators, Targets and Interconnect

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

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, Doulos, 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

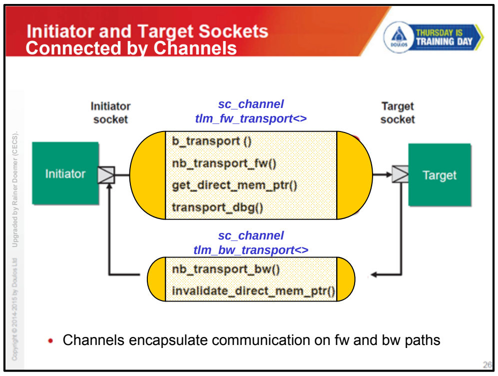


Initiator and Target Sockets

- Sockets provide fw and bw paths, and group interfaces

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, Doulos, 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



Initiator and Target Sockets Connected by Channels

- Channels encapsulate communication on fw and bw paths

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)

Concluding Remarks

- Towards *standard-compliant parallel* SystemC
  - Higher simulation speed on multi/many core hosts
- Overcome the identified IEEE 1666-2011 obstacles
  - Move up from DES to PDES
  - Adopt a parallel mindset, expose and exploit parallelism
  - Apply the principle of separation of concerns
    - Modules encapsulate computation
    - Channels encapsulate communication
  - Simulate models faster with parallel execution semantics
- ➢ SystemC must evolve in a major revision (3.x)
  - C++11 already has built-in support for multithreading
  - SystemC must embrace true parallelism
    - Otherwise it will go down the same path as the dinosaurs…

Acknowledgments

- For helpful input, fruitful discussions, and honest feedback, I would like to thank:
  - Tim Schmidt
  - Guantao Liu
  - Desmond Kirkpatrick
  - Abhijit Davare
  - Ajit Dingankar
  - Philipp Hartmann

  - and all participants in the SystemC Evolution Day 2016
    - …for not kicking me off the stage! ☺