# Seven Obstacles in the Way of Standard-Compliant Parallel SystemC Simulation

Rainer Dömer, *Member, IEEE*

***Abstract*— The IEEE 1666-2011 standard defines SystemC based on traditional discrete event simulation and sequential co-routine semantics, despite explicit parallelism in the model and ample parallel processor cores available in today's host computers. In order to evolve the SystemC standard towards faster *parallel discrete event simulation*, substantial hurdles must be overcome. This letter identifies seven obstacles in the standard that stand in the way of efficient parallel SystemC simulation, namely the co-routine semantics, simulator state, lack of thread safety, the role of channels, TLM-2.0, sequential mindset, and temporal decoupling. For each obstacle, we discuss the problem and propose a potential solution toward truly parallel SystemC. This letter to the editor is meant to identify difficulties with IEEE SystemC and stimulate fruitful discussion in the community.**

***Index Terms*—Discrete event simulation, multithreading, parallel discrete event simulation, parallel processing, simulation, SystemC, system level description language, system level design.**

## I. Introduction

THE SystemC language [1] defines its execution semantics based on traditional *discrete event simulation (DES)* where a central scheduler manages a set of concurrent threads driven by events and simulation time advances. As a consequence, SystemC simulation is generally subject to partial temporal ordering of the threads with barriers (delta and time cycles). Specifically, the SystemC standard IEEE 1666-2011 [2] requires *cooperative multi-tasking* semantics where only a single thread is active at any time. Following this, most simulators, including the open source proof-of-concept library [3], implement fully sequential execution which cannot exploit the parallelism exhibited by the model. Since highest simulation speed is critical due to the rising system complexity, *parallel discrete event simulation (PDES)* [4] is very desirable as it maintains the level of abstraction and executes threads at the same simulation time in parallel and thus can utilize multiple processor cores available on the host computer and speed up the simulation nearly linearly [7][12], or even super-linearly [6]. In other words, hours of simulator runtime can be reduced to minutes.

Manuscript received May 19, revised August 15, accepted October 10, 2016. Date of publication TBD. This work was supported in part by Intel Corporation for the project "Out-of-Order Parallel Simulation of SystemC Virtual Platforms on Many-Core Architectures".

Rainer Dömer is a Senior Visiting Fellow at the University of New South Wales, Sydney 2052, Australia, and an Associate Professor at the Center for Embedded and Cyber-Physical Systems at the University of California, Irvine, CA 92697, USA (e-mail: doemer@uci.edu).

Unfortunately, the current IEEE 1666-2011 standard imposes significant restrictions on PDES for SystemC. Whereas proposed parallel SystemC approaches largely ignore the strict rules of the standard, e.g. [5-7], this letter analyzes the problem of *standard-compliant* parallel SystemC simulation, identifies seven obstacles[1] in the standard language reference manual (LRM), and outlines possible solutions including changes to the standard for the next generation of SystemC. As such, this letter (and its controversial content) is intended to start a discussion towards a major revision of the SystemC standard suitable for PDES. To this end, we contribute a technical review and evaluation of the SystemC LRM [2] and corresponding proof-of-concept library version 2.3.1 [3].

## II. Obstacle 1: Co-Routine Semantics

The first obstacle in the way of truly parallel simulation is the fact that the standard LRM explicitly specifies *"co-routine semantics"* also known as *"co-operative multitasking"*. LRM Section 4.2.1.2 requires that during the evaluation phase *"only a single process instance can be running at any one time"* and the *"scheduler is not pre-emptive"* ([2], pp. 17, 18).

### A. Problem: Uninterrupted execution guarantee

The SystemC LRM explicitly outlines the restriction to non-preemption as it applies to multi-core parallel execution: *"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"*. The problem here is illustrated in Fig. 1 where two threads access a shared variable `x`. The required non-preemptive execution guarantees that `thread1` and `thread2` safely output `1` and `42`, respectively. However, parallel execution with possible preemption of the two threads is *not* safe due to the race condition around variable `x`. Here, the LRM requires that such conflicting parallel accesses to shared variables are prevented by the simulator. This is automatic (comes for free) in a sequential implementation, but is difficult to achieve in a parallel approach. To be standard-compliant, a PDES simulator must *"analyze any dependencies"* among all threads and *"constrain their execution to match the co-routine semantics"* ([2], p. 18). Whereas this is feasible, for example by use of advanced static compiler analysis [8,9], it places an undue burden on the simulator which then requires the use of a

[1] The seven obstacles are not independent and listed in no particular order. We also make no claim of completeness of the identified problems.

```
1   int x;  // shared global variable
2
3   void thread1()      void thread2()
4   { x = 0;            { x = 7;
5     x = x + 1;          x = x * 6;
6     std::cout << x;     std::cout << x;
7   }                   }
```

Fig. 1. Example of two conflicting SystemC threads: Current co-routine semantics guarantee safe output of 1 by thread1 and 42 by thread2.
In contrast, a parallel execution results in a race condition around the shared variable x with undefined behavior.

```
1   void thread1()      void thread2()
2   { int x = 0;        { int x = 7;
3     x = x + 1;          x = x * 6;
4     std::cout << x;     std::cout << x;
5   }                   }
```

Fig. 2. Example of two conflict-free SystemC threads: Local variables result in thread safe execution under both the current co-routine semantics as well as the proposed parallel execution semantics.

dedicated SystemC-aware compiler (instead of GNU C++).

### B. Proposal: Assume parallel execution (with preemption)

To avoid such unnecessary complexity, we propose to explicitly specify parallel execution semantics for SystemC, including the implication of possible preemption. If the LRM states from the beginning (i.e. in Section 4 on simulation semantics) that process instances may execute in parallel when they are at the same simulation time (same delta and time cycle), then we can model naturally parallel designs with truly parallel semantics and simulate them with faster parallel execution. The only caveat is then that the model designer must pay attention to parallel programming and write thread safe code without race conditions, as shown in Fig. 2.

```
1   bool sc_pending_activity_at_current_time();
2   bool sc_pending_activity_at_future_time();
3   bool sc_pending_activity();
4   sc_time sc_time_to_pending_activity();
```

Fig. 3. Example of SystemC API functions presuming DES state ([2], p. 31).

## III. Obstacle 2: Simulator State

The second obstacle in the SystemC LRM is the fact that DES is presumed in the simulator application programming interface (API). As an example, Fig. 3 shows four functions that expose the internal state of the simulator to the user.

### A. Problem: PDES is different from sequential DES

The problem here is that the desired PDES is inherently different from the presumed DES. For instance, after elaboration there may be multiple threads running in parallel and scheduling may occur while other threads are still active.

### B. Proposal: Revise simulator state API for PDES

To address this mismatch, we propose to carefully review the simulator state primitives and the associated semantics, and revise both appropriately for PDES. Specifically, the functions shown in Fig. 3 (and other similar APIs) need to be adapted for a parallel scheduler. As an example, one could add a new `sc_parallel_activity()` indicating concurrent activity.
Note that the general notion of *shared state* requires careful consideration when moving from DES to PDES. Whereas the simulator state is visible to the user (Fig. 3), other shared state is hidden in the SystemC library, as the next obstacle shows.

```
1   sc_length_param   length10(10);
2   sc_length_context cntxt10(length10);
3   sc_int_base       int_array[2];
```

Fig. 4. Example of SystemC sequential shared state ([2], p. 194): a length-10 parameter is constructed, then a context with this parameter is created, and finally an array of 10-bit integers is defined using the *current* context.

```
1   template <class T> inline
2   void sc_fifo<T>::write( const T& val_ )
3   {  sc_stacked_lock l(m_mutex); // new channel lock
4      while( num_free() == 0 ) {
5          sc_core::wait( m_data_read_event );
6      }
7      m_num_written ++;
8      buf_write( val_ );
9      request_update();
10  }
```

Fig. 5. Example of thread safe communication: The blocking write method in the primitive channel sc_fifo ([3], header file sc_fifo.h) is protected by a proposed sc_stacked_lock (line 3) which automatically locks the channel instance on entry (by acquiring a mutex provided in the channel base class) and unlocks the channel instance on exit (by releasing the mutex again).
Note that without the added sc_stacked_lock (line 3), there would be a race condition between the shared variables m_num_written and num_free.

## IV. Obstacle 3: Lack of Thread Safety

Generally, SystemC primitives are not multi-thread safe. In fact, the LRM requires thread safety only for a single function, namely `async_request_update` ([2], p. 121) and the proof-of-concept library [3] implements many SystemC primitives with shared state which is safe only under sequential DES. Fig. 4 shows a suspicious[2] example fragment where variables are defined based on a prior established context (line 2). If another thread creates a different context in parallel, then it is undefined which context is applied at the time of the variable definition (line 3).

### A. Problem: Parallel execution may lead to race conditions

This problem is again well-known as a race condition which must be prevented because it otherwise results in undefined behavior. Here parallel updates to the shared context need to be properly synchronized, for example, by atomic operations or explicit locks (binary semaphores). However, identifying such critical regions in the code is difficult for the user who is unaware of the actual implementation in the SystemC library.

### B. Proposal: Require all primitives to be multi-thread safe

To resolve this problem, we propose for the LRM to require that *all* SystemC primitives shall be implemented in a multi-thread safe manner (i.e. add this requirement to Section 3.3 next to the discussion on side-effects). Following this, the proof-of-concept library must be carefully reviewed and revised accordingly (which arguably is significant work).

## V. Obstacle 4: Class sc_channel

The fourth obstacle in the way of standard-compliant parallel SystemC appears at first sight only as a small technicality, but that has significant impact on safe communication under

[2] In the proof-of-concept library [3], the class `sc_context` is commented as "co-routine safe" only. To make this example multithread safe, thread local storage would be needed in the implementation.

PDES. The LRM specifies that *"typedefs sc_behavior and sc_channel are provided for users to express their intent"* ([2] p. 56) and the proof-of-concept library [3] accordingly implements a `typedef sc_module sc_channel` in the header file sc_module.h. Thus, sc_channel is in fact only an alias type for sc_module.

### A. Problem: sc_channel appears identical to sc_module

In the C++ language, which SystemC is based on, a typedef is only another name, not a new type. Thus, sc_module and sc_channel are the same for any compiler or synthesis tool and cannot be distinguished. In other words, there is no sc_channel. This breaks a key system design principle, namely the clear *separation of computation and communication* [10] also known as the *orthogonalization of concerns* [11].

The separation of communication and computation is critical in PDES because computation code stays clear of shared variables (to allow fast and safe parallel execution), but sharing cannot be avoided in communication between threads. Communication methods naturally rely on shared variables and events, and parallel accesses to those must be properly synchronized.

Fig. 5 lists the blocking write method of the primitive channel sc_fifo as an example, where a race condition is prevented by a newly introduced channel lock (line 3). Note that this synchronization is necessary for safe communication, but unwanted for computation. Thus, we need to separate the two by clearly distinguishing channels from modules.

### B. Proposal: Class sc_channel derived from sc_module

We propose to resolve this obstacle by replacing the type alias with a uniquely identifiable type, specifically by a proper class sc_channel that is derived from sc_module. While this creates distinguishable types for channels and modules, it allows at the same time the sharing of common features (e.g. object name). Then the channel base class can also provide the sc_stacked_lock member suggested in Fig. 5 which is not needed in modules.

Most importantly, this change reinstates the system design principle of separation of concerns for SystemC. The modules encapsulate the computation (host active threads/processes) and the channels encapsulate the communication (implement the interface methods) in a truly parallel design model.

## VI. Obstacle 5: TLM-2.0

The proposal of using the channel as a *monitor* with access synchronization cleanly resolves Obstacle 4 and reestablishes thread safe communication, but leaves open the next obstacle, namely TLM-2.0 [2]. As shown in Fig. 6, communication between initiator and target modules follows well-defined interfaces in TLM-2.0 (ensuring the interoperability of components from different sources), but there is no channel to encapsulate the communication methods.

### A. Problem: Channel concept has disappeared

In contrast to TLM-1.0 where a channel wraps the message passing communication, TLM-2.0 uses references and pointers

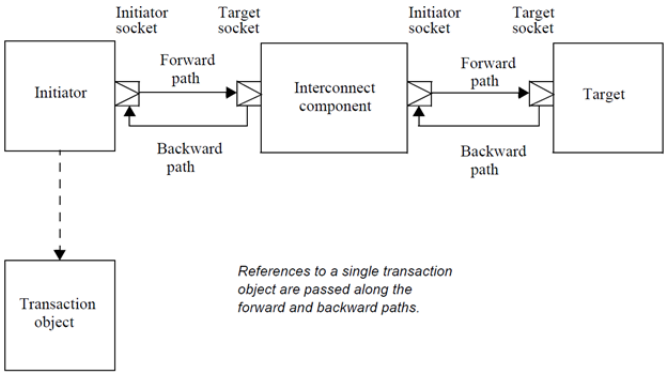


Fig. 6. TLM-2.0 communication between initiators and targets ([2], p. 421): A transaction object is passed by reference through sockets and interconnect components along forward and backward paths. Whereas interface methods are well-defined, there is no channel to encapsulate the communication.

for direct memory access into other modules, including a direct memory interface (DMI). The interface methods are implemented directly in the modules without containment constructs that could offer multithread access synchronization.

### B. Proposal: Encapsulate communication in channels

While further and thorough study of possible protection schemes is needed here, we propose the conceptual solution of wrapping the TLM-2.0 communication methods into actual channels (similar to TLM-1.0) so that the same protection with locks as proposed for Obstacle 4 can be applied here as well. For safe DMI access, atomic type operations may prove useful as well, since this could result in lock-less synchronization.

## VII. Obstacle 6: Sequential Mindset

Probably the biggest obstacle in the quest towards truly parallel SystemC is the sequential modeling mindset that SystemC designers are used to. A good example is the fact that SC_METHOD is preferred over SC_THREAD because thread context switches are considered overhead: *"context switching between thread processes may impose a simulation overhead when compared with method processes"* ([2], p. 44).

### A. Problem: Sequential modeling is encouraged

The difficult challenge is the heavy bias towards sequential SC_METHODs, where true threads with parallel context (own execution stack) are avoided for the sake of saving a few cycles in sequential DES. Avoiding context switches is the wrong optimization criterion. Simulation speed will be much more improved by parallel execution. Thus, the designer's efforts should be focused on exposing parallelism in the model as much as possible so that that can be exploited in both simulation and model implementation.

### B. Proposal: Encourage parallel modeling with true threads

The targeted systems are parallel by nature, so should be their models. We propose to strongly promote a parallel modeling mindset toward true thread-level parallelism (which arguably requires rethinking and retraining). In PDES, there is no need for SC_METHOD anymore (since context switches are of a different kind), so SC_METHOD can actually be eliminated (which also avoids complexity of next_trigger() vs. wait()). True threads should reflect the naturally parallel system's

behavior and the observed task relations should be explicitly expressed using synchronization primitives (**`event.notify`**, **`wait(event)`**) and communication (channel) constructs.

## VIII. Obstacle 7: Temporal Decoupling

As defined in the LRM ([2], p. 453), *temporal decoupling* (TD) allows SystemC threads to *"run ahead of the simulation time for an amount of time known as the time quantum"* in order to improve the simulation speed *"by reducing the number of context switches and events"*. Again, context switches are identified as a main impediment to simulation speed. Whereas the trade-off of accuracy for speed fits the well-known mechanism of abstraction in system level design, SystemC TD and its "*global quantum"* ([2], p. 453) are designed specifically for sequential DES.

### A. Problem: PDES is a different foundation than DES

There are two problems with this TD when moving to PDES. First, the global time quantum (a singleton) is a technical obstacle, because it directly leads to a race condition for parallel threads which would need to be prevented by synchronization, defeating its very purpose.

Second, sequential and parallel DES are very different foundations. The sequential assumptions SystemC TD was designed for, do not hold true anymore under PDES. Context switches in PDES are of a different nature and therefore the current TD is incompatible with parallel simulation.

### B. Proposal: Reevaluate temporal decoupling for PDES

To overcome this obstacle, we propose to redesign and reevaluate the idea of TD for PDES (despite the cost of repeating some of the valuable TD research to date). Conceptually, TD and parallel execution should be orthogonal, both independently providing higher execution speed. Specifically, we advocate for a true **`wait(time)`** handled by the parallel SystemC kernel, instead of the global quantum managed by the user (agnostic to the kernel). Here, the use of modern compiler techniques may prove useful to automatically optimize timing and parallel scheduling [8].

## IX. Conclusion

Moving up from DES to PDES semantics will allow improved simulation speed on multi and many core hosts by an order of magnitude. For SystemC, however, significant difficulties are imposed by the current IEEE 1666-2011 standard.

In this letter, we have identified seven obstacles in the way of standard-compliant parallel SystemC simulation. In order to overcome these identified obstacles, we need to adopt a parallel modeling mindset so that the natural parallelism in the target system is exposed in the model and can be efficiently exploited. We must apply the system design principle of separation of concerns and clearly encapsulate communication in channels and computation in modules. Consequently, the IEEE SystemC language standard must evolve in a major revision (similar to C++11 which recently has accomplished built-in support for multithreading).

We advocate for the next generation of the SystemC standard to embrace true parallel simulation and offer the analysis provided in this letter as a starting point for discussion.

## Acknowledgment

For helpful input, fruitful discussions, and honest feedback, the author would like to thank A. Davare, A. Dingankar, P. Hartmann, D. Kirkpatrick, G. Liu, and T. Schmidt, as well as the anonymous reviewers and all participants in the SystemC Evolution Day 2016 in Munich, Germany.

This work has been supported in part by funding from Intel Corporation. The author thanks Intel Corporation for the valuable support.

## References

[1] T. Grötker, S. Liao, G. Martin, S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[2] IEEE Computer Society. *IEEE Standard 1666-2011 for Standard SystemC Language Reference Manual*, IEEE, New York, USA, 2011.

[3] SystemC Language Working Group. *SystemC 2.3.1, Core SystemC Language and Examples*, Accellera Systems Initiative, USA, 2014. http://accellera.org/downloads/standards/systemc

[4] R. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, Oct 1990.

[5] P. Ezudheen, P. Chandran, J. Chandra, B. Simon, D. Ravi. "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines". In Proc. PADS, pp. 80–87, 2009.

[6] C. Schumacher, R. Leupers, D. Petras, A. Hoffmann. „parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures". In Proc. CODES+ISSS, pp. 241–246, 2010.

[7] R. Sinha, A. Prakash, H. Patel. Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs. In Proc. ASPDAC, Sydney, Australia, 2012.

[8] W. Chen, X. Han, C. Chang, G. Liu, R. Dömer. Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models. *IEEE TCAD*, 33(12):1859–1872, Dec. 2014.

[9] G. Liu, T. Schmidt, R. Dömer. *RISC Compiler and Simulator, Alpha Release V0.2.1: Out-of-Order Parallel Simulatable SystemC Subset*, TR CECS 15-02, CECS, UC Irvine, USA, 2015.

[10] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[11] K. Keutzer, A. Newton, J. Rabaey, A. Sangiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-based Design. IEEE TCAD, 19(12):1523-1543, Dec. 2000.

[12] W. Chen, X. Han, C. W. Chang, R. Dömer, Advances in Parallel Discrete Event Simulation for Electronic System-Level Design. *IEEE Design & Test*, vol. 30, no. 1, pp. 45-54, Feb. 2013.

**Rainer Dömer** (S'96–M'00) received the Ph.D. degree in information and computer science from the University of Dortmund, Dortmund, Germany, in 2000.

He is currently a Senior Visiting Fellow at the University of New South Wales, Sydney 2052, Australia, and an Associate Professor with the Electrical Engineering and Computer Science Department at the University of California, Irvine, CA, USA, where he is a faculty member of the Center for Embedded and Cyber-Physical Computer Systems.

Prof. Dömer's research interests include system-level design and methodologies, embedded and cyber-physical computer systems, specification and modeling languages, and advanced parallel discrete event simulation.