Project Title: Parallel Simulation of SystemC Loosely-Timed Transaction Level Models

KONSTANTINOS SOTIROPOULOS
konstantinos.sotiropoulos@intel.com

February 18, 2016

Contents

1 Acronyms

DE: Discrete Event
DES: Discrete Event Simulator/Simulation
DMI: Direct Memory Interface
ESL: Electronic System-Level Design
HPC: High Performance Computing
MoC: Model of Computation
MPSoC: Multicore System on Chips
OoO: Out-of-Order
PDES: Parallel Discrete Event Simulation
SLDL: System-Level Design Language
SMP: Symmetric Multiprocessing
SoC: System on Chip
SR: Synchronous Reactive
TLM: Transaction Level Modeling
CMB: Chandy/Misra/Bryant algorithm

2 Organization

This is a Master's Thesis project that will be carried out in Intel Sweden AB and is supervised by KTH's ICT department. Mr. Bjorn Runaker (bjorn.runaker@intel.com) is the project's supervisor from the company's side, while professor Ingo Sanders (ingo@kth.se) and PhD student George Ungureanu (ugeorge@kth.se)are the examiner and supervisor from KTH. The project begun on 2016-01-16 and will finish on 2016-06-30, as dictated by the contract of employment that I, Konstantinos Sotiropoulos a Master's student at the Embedded Systems program, have signed with the company (document title: "Statement of Terms and Conditions of Fixed Term Employment").

The scope of this project has been and is being mutually determined by all parties. It is dialectically determined between the company's needs and the institute's research agenda. As Master's Thesis project, it needs to expose a scientific ground on which the engineering effort shall be rooted.

All the necessary equipment (software and hardware) has been kindly provided by the company. The exact legal context that will apply to any software produced as a result of this project is yet to be determined, but will conform to the general context dictated by the documents already signed (documents' titles: "Statement of Terms and Conditions of Fixed Term Employment" and "Employee Agreement") .

3 Background

SystemC is considered to be a System-Level Design Language (SLDL), that is implemented as a C++ class library. The language is maintained and promoted by Accellera (former Open SystemC Initiative OSCI) and has been standardized with the latest LRB being the IEEE 1666-2011 [1] The major advantage SystemC provides to the designer is that a component's functionality can be easily expressed in C/C++, the dominating languages for in systems programming (e.g. Operating System and Device Driver development). As a result SystemC models are executable: the designer can directly simulate the behavior of a system modeled in SystemC. The language has been demanded and supported by the semiconductor industry, in its collaborative effort to push Electronic Design Automation (EDA) into higher abstraction layers, thus giving birth to a trend referred to as Electronic System-Level Design (ESLD).

The rest of this section is devoted into presenting the theoretical/scientific ground on which we will cultivate this project. To avoid confusion by being precise, fundamental terms and concepts of computer science are being defined and redefined, even at the risk of being pedantic. Facing a labyrinth of abstraction layers, we will begin unraveling Ariadne's thread at the simple observation that SystemC is a System Level Design **Language**.

3.1 Models of Computation

A **language** is a set of symbols, rules for combining them (its syntax), and rules for interpreting combinations of symbols (its semantics). Two approaches to semantics have evolved: denotational and operational. Operational semantics, which dates back to Turing machines, gives the meaning of a language in terms of actions taken by some abstract machine. How the abstract machine in an operational semantics can behave is a feature of what we call the **Model of Computation (MoC)** [2]. This definition implies that languages are not computational models themselves, but have underlying computational models [3].

How can this definition make sense for SLDLs and what do we expect a SLDL to be able to do? A SLDL should be able to describe a system as a hierarchical network of interacting components. A MoC is therefore a collection of rules to define what constitutes a component and what are the semantics of execution, communication and concurrency of the abstract machine that will execute the model [3, 4]. To ensure meaningful simulations the MoC of the abstract machine that simulates a model must be equivalent with that of the abstract machine that will realize the system.

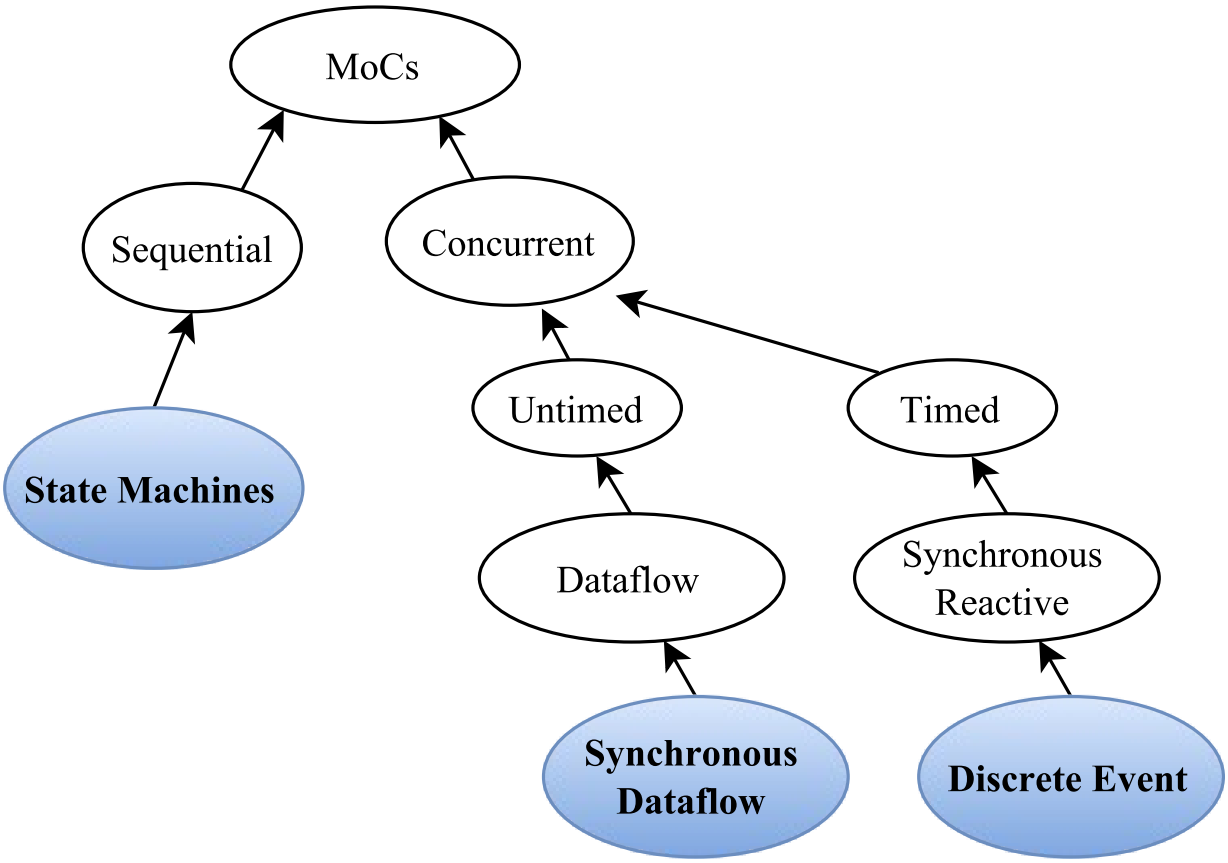

Figure 1: Categorization of three of the most explored MoCs: State Machine, Synchronous Dataflow and Discrete Event(adopted from [4])

3.2 The Discrete Event Model of Computation

The dominant MoC that underlies most industry standard EDA languages (VHDL, Verilog, SystemC) is called **Discrete Event (DE)**. The components of a DE system are called **processes**. In this context processes usually model the behavior and functionality of hardware entities. The execution of processes is concurrent and the communication is achieved through **events**. An event can be considered as an exchange of a time-stamped value.

Concurrent execution does not imply parallel/simultaneous execution. The notion of **concurrency** is more abstract. It can be realized as either parallel/simultaneous execution or as sequential interleaved execution, depending on the actual machine's computational resources.

Systems whose semantics are meant to be interpreted by a DE MoC, in order to be realizable, must have a **causal** behavior: they must process events in a chronological order, while any output events produced by a process are required to be no earlier in time than the input events that were consumed [4]. At any moment in real time, the model's time is determined by the last event processed.

In figure 1 one can observe that the DE MoC is also considered to be **Synchronous-Reactive (SR)**. This demonstrates the possibility of the MoC to "understand" entities with zero execution time, where output events are produced at the same time input events are consumed. We can also extend/rephrase the previous definitions and say that Synchronous-Reactive MoCs are able to handle systems where events happen at the same time, instantaneously, in a causal way. The DE MoC handles the aforementioned situations by extending time-stamps(the notion of model time) with the introduction of delta delays (also referred to as cycles or micro-steps). A delta delay signifies an infinitesimal unit of time and no amount of delta delays, if summed, can result in time progression. A time-stamp is therefore represented as a tuple of values, (t, n) where t indicates the model time and n the number of delta delays that have advanced at t.

3.3 The Discrete Event Simulation(or)

A realization of the DE abstract machine is called a **Discrete Event Simulator (DES)**. SystemC's reference implementation of the DES is referred to as the **SystemC kernel** [1].

Concurrency of the system's processes is achieved through the co-routine mechanism (also known as co-operative multitasking). Processes execute without interruption. In a single core machine that means that only a single process can be running at any real time, and no other process instance can execute until the currently executing process instance has yielded control to the kernel. A process shall not preempt or interrupt the execution of another process [1].

To avoid quantization errors and the non-uniform distribution of floating point values, time is expressed as an integer multiple of a real value referred to as the time resolution.

The kernel maintains a **centralized event queue** that is sorted by time-stamp and knows which process is **running**, which processes are waiting for events and which are **runnable**. Runnable processes have had events to which they are sensitive triggered and are waiting for the running process to yield to the kernel so that they can be scheduled. The kernel controls the execution order by selecting the earliest event in the event queue and making its time-stamp the current simulation time. It then determines the process the event is destined for, and finds all other events in the event queue with the same time-stamp that are destined for the same process [5]. The operation of the kernel is exemplified in Alg 1.

Algorithm 1 SystemC event loop, adopted from [6]

```
1:  while timed events to process exist do                ▷ Simulation time progression
2:      trigger events at that time
3:      while runnable processes exist do                 ▷ Delta cycle progression
4:          while runnable processes exist do
5:              run all triggered processes
6:              trigger all immediate notifications
7:          end while
8:          update values of changed channels
9:          trigger all delta time events
10:     end while
11:     advance time to next event time
12: end while
```

3.4 The Parallel Discrete Event Simulation(or)

The previous section has made evident that the reference implementation of the SystemC kernel assumes sequential execution and therefore can not utilize modern massively parallel host platforms. The most logical step in achieving faster simulations is to realize concurrency, from interleaving process execution to actual simultaneous/parallel execution. By assigning each process to a different processing unit of the host platform (core or hardware thread) we enter the domain of **Parallel Discrete Event Simulation (PDES)**. SystemC as a SLDL remains the same while the implementation of the DES is radically different.

By allowing processes to execute simultaneously one can allow each process to have its own perception of simulation time, determined by the last event it received. This approach is referred to as **Out-of-Order PDES (OoO PDES)** [7]. Examples of OoO PDES simulators are the SystemC-SMP [8] and SpecC [9], although the latter is not meant for SystemC.

For PDES implementations that enforce global simulation time, the term Synchronous PDES has been coined in the parSC simulator[6]. In Synchronous PDES, parallel execution of processes is performed within a delta cycle. With respect to Alg 1, we can say that a Synchronous PDES parallelizes the execution of the innermost loop (line 4). However, as we will see in the following section, this approach will bare no fruits in the simulation of TLM Loosely Timed simulations, since delta cycles are rarely triggered [10].

Finally, before committing into modifying the SystemC DES, we should mention the existence of less intrusive approaches, that instead of redesigning extend the reference kernel. The example of the sc-during

SystemC library [11] is characteristic. To exploit parallelism, each process must be redefined as a sequence of atomic tasks that have duration (in simulation time). The term atomic is used to represent the fact that these tasks are insensitive to input/output events for their duration. Thus, the kernel can safely assign them to a different operating system thread and allow them to execute independently from the rest of the simulation.

3.5 Transaction Level Modeling

Transaction Level Modeling (TLM) enhances SystemC's expressiveness in order to facilitate the more abstract description (compared to RTL) of systems. TLM 2.0 allows model interoperability and the rapid development of fast virtual platforms to be deployed in software development, early on in the system's design procedure.

Transaction-level models represent one specific type of the DE MoC [12]. Transactions are non-atomic communications, normally with bidirectional data transfer, and consist of a set of messages that are usually modeled as atomic communications. The set of messages exchanged during a transaction are modeled using function calls and represent the different phases of a communication protocol. In a transaction one can distinguish two actors: the **initiator**, the process which initiated the communication, and the **target**, the process which is supposed to service the target's request.

TLM 2.0 API [13] consists of the following features:

- A set of core interfaces
  - A Blocking interface which is coupled with the **Loosely-Timed (LT)** coding style.
  - A non-blocking interface, which is coupled with the **Approximately-Timed (AT)** coding style.
  - The **Direct Memory Interface (DMI)** to enable an initiator to have direct access to a target's memory, bypassing the usual path through the interconnect components used by the transport interfaces.
  - The **Debug transport interface** to allow an non-intrusive inspection of the system's state.
- The **global quantum** used by the **temporal decoupling** mechanism of the LT coding style, which facilitates faster simulations by reducing the number of context switches performed by the kernel.
- Initiator and target **sockets** to denote the links (causal dependencies) between processes.
- The **generic payload** which supports the abstract modeling of memory-mapped buses.
- A set of **utilities** to facilitate the rapid development of models.

Figure demonstrates the typical use cases for TLM's different features.

4 Problem statement

The distribution of simulation time opens Pandora's box. Protecting the OoO PDES from **causality errors** demands certain assumptions and the addition of complex implementation mechanisms.

The first source of causality errors arises when the system's state variables are not distributed, in a disjoint way, among the processes [14]. A trivial realization of the above scenario is depicted in figure 3. Processes P_1 and P_2 are executing simultaneously, while sharing the system's state variable x. Events E_1 and E_2 are executed by P_1 and P_2 respectively. If we assume that in real time E_2 is executed before E_1, then we have implicitly broken causality, since E_1 might be influenced by the value of x that the execution of E_2 might have modified. Furthermore, one must observe that this kind of implicit interaction between P_1 and P_2 can not be expressed in a DE MoC. This is a meta-implication of the host platform's shared memory architecture.

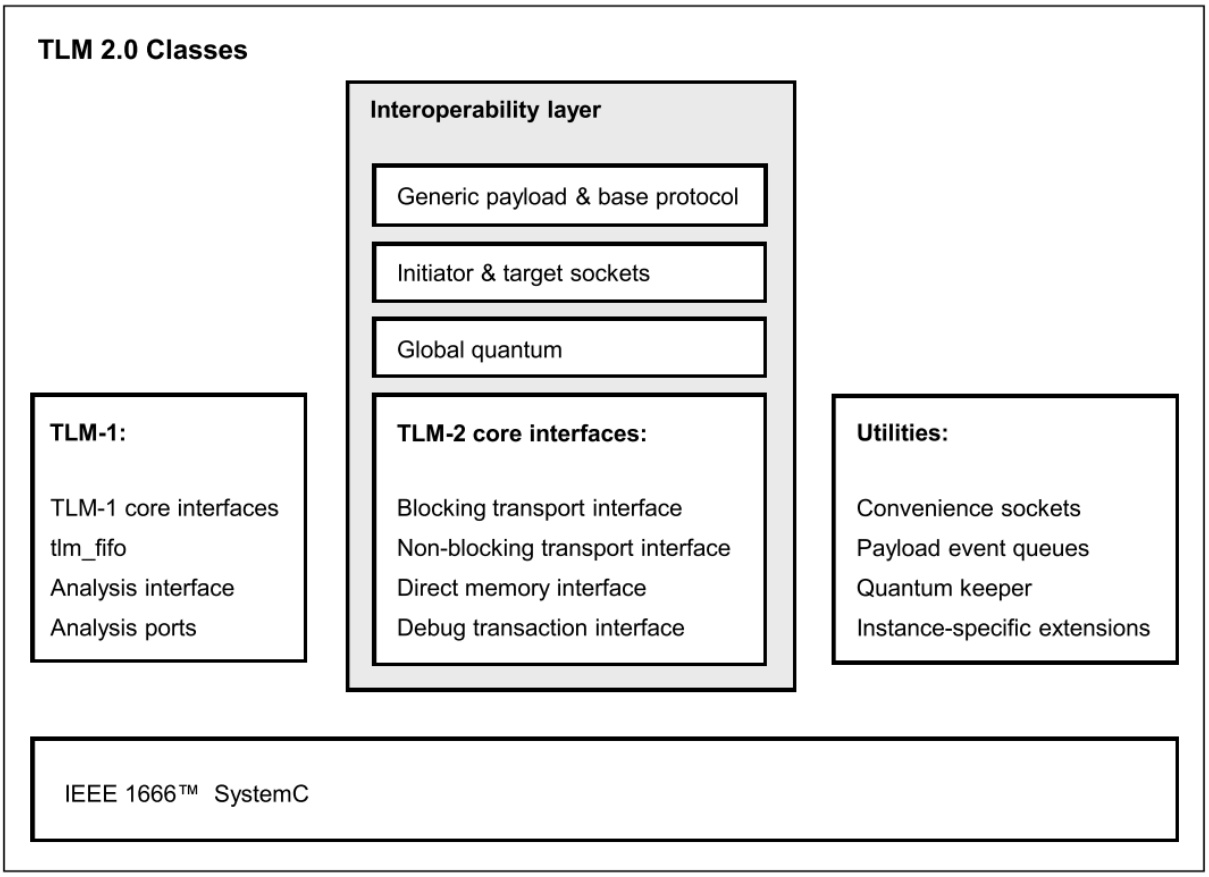

Figure 2: TLM 2.0 use cases (adopted from [13]).

The second and most difficult to deal with source of causality errors is depicted in figure 4. Event E_1 affects E_2 by scheduling a third event E_3 which, for the shake of argument, modifies the state of P_2. This scenario necessitates sequential execution of all three events. Thus the fundamental problem in PDES, in the context of this scenario, becomes the question: how can we deduce that it is safe to execute E_2 in parallel with E_1, without actually executing E_1 [14]? However, one must notice that the kind of interaction that yields this problematic situation is explicitly stated in the model.

The last example makes evident the fact that the daunting task of preserving causality in the simulation is all about **process synchronization**. For example, each process must be able to communicate to each of its peers (processes that is linked with) the message: "I will not send you any event before t_1, so you can proceed with processing any event you have with time-stamp t_2 where $t_2 < t_1$".

PDES synchronization algorithms, with respect to how they deal with causality errors, have been classified into two categories: **conservative** and **optimistic** [15]. Conservative mechanisms strictly avoid the possibility of any causality error ever occurring by means of model introspection and static analysis. On the other hand, optimistic/speculative approaches use a detection and recovery approach: when causality errors are detected a rollback mechanism is invoked to restore the system. An optimistic compared to a conservative approach will theoretically yield better performance in models where communication, thus the probability of causality errors, is below a certain threshold [14].

Both groups present severe implementation difficulties. For conservative algorithms, model introspection and static analysis tools might be very difficult to develop, while the rollback mechanism of an optimistic algorithm may require complex entities, such as a hardware/software transactional memory [16] .

5 Problem

In this project we investigate the feasibility of implementing a SystemC OoO PDES, that can lead to scalable simulations of MPSoC Loosely-Timed Transaction Level Models, on SMP host platforms.

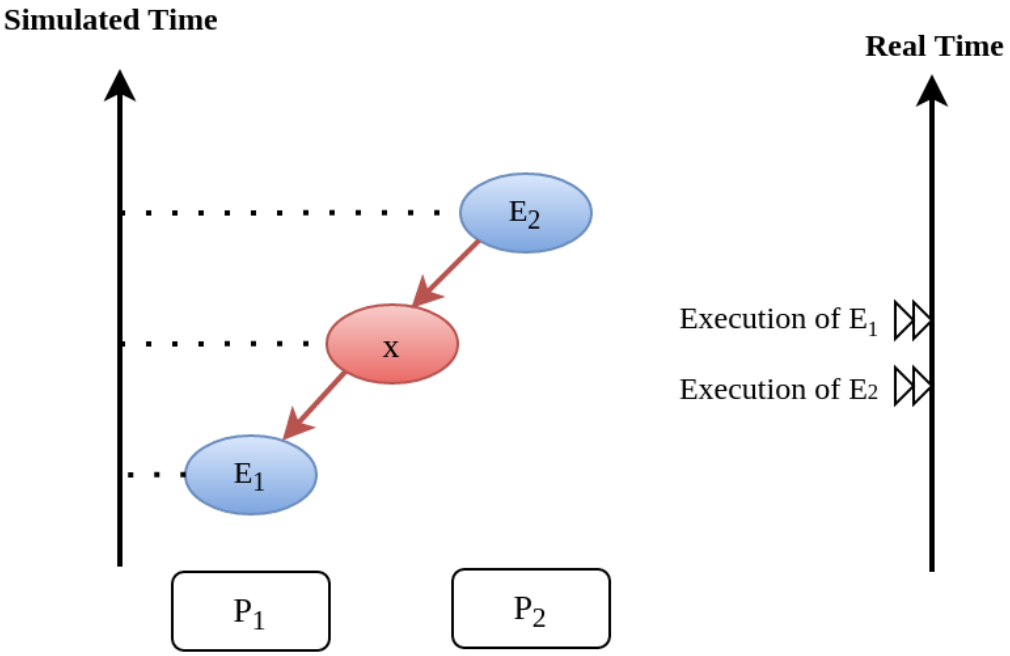

Figure 3: Causality error caused by the sharing of the system's state variable x by P_1 and P_2.

6 Hypothesis

We hypothesize that by following a conservative approach on implementing a SystemC OoO PDES we will yield semantically equivalent and scalable simulations with respect to the reference SystemC DES.

7 Purpose

The vision of a fully automated and connected society, the IoT revolution has promised to deliver, is depending on the industry's ablate to deliver novel, complex and heterogeneous cyber-physical systems with short time-to-market constraints. To live up to these expectations, the engineering discipline of ESLD must provide an answer to a number of questions:
High-Level Synthesis: How a system described in a SLDL can be realized in a structured and automatic way? Which of its components should be mapped in hardware entities like Digital Signal Processors (DSPs), Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs)? Which of its components could be software for some kind of Central Processing Unit (CPU)? Which is the optimal mapping that satisfies the system's requirements and yields a minimum power consumption?
Correct-by-design: Can a high-level synthesis design methodology yield correct-by-design implementations? Can a system be formally verified given its abstract representation, early on in the design procedure? Can we free the huge amount of resources wasted in mundane testing and debugging procedures that sometimes can not even provide any formal guarantee about the system's behavior?
Improving the co-simulation speed for hardware and software: Can we develop a virtual prototype of the platform early on in the development cycle, so that software engineers can begin developing integral applications without having to wait for the silicon to arrive? Can we make the simulation fast and accurate, utilizing all the latest developments in High Performance Computing (HPC)? This project hopes to deliver an infinitesimal contribution in solving the latter class of questions.

8 Goals and Objectives

If the timing constraints stretched beyond the scope of a Master Thesis, the project's self-actualization would require the development/production of the following components (sorted in descending significance

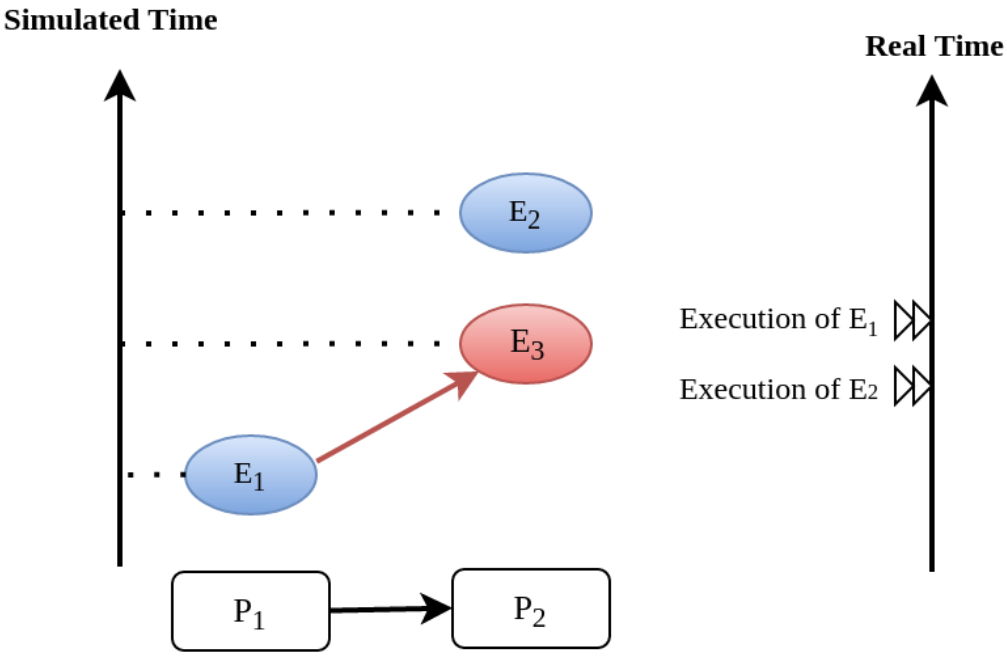

Figure 4: Causality error caused by the unsafe execution of event E_2 (adopted from [14]).

order):

1. An OoO PDES implementation of the SystemC kernel.
2. A proof of concept application of the proposed kernel, on a sufficiently parallel system, running a substantially parallel application, on top of a Linux kernel.
3. The Master Thesis report document.
4. A static analysis/introspection tool for parsing the SystemC description of the system and extracting its pure representation, in terms of processes and links.
5. A code generation tool for constructing the communication and synchronization mechanisms.
6. A way of sequencing the application of the previous tools, either in the kernel's elaboration phase, or using a "gluing" script.
7. A TLM 2.0 coding style to minimize the effort and complexity of the analysis and generation tools.
8. A way to ensure the kernel's OSCI compliance.
9. A roadmap for elevating the simulation from SMP parallel to distributed, in a cluster of SMP nodes, parallel.

Given the time constraints, the primary focus falls on the first three objectives. The automation and generality the tools could deliver will be emulated by manual and ad-hoc solutions.

9 Methods

9.1 Assumptions and delimitations

The IEEE Standard for SystemC states the following about non reference implementations of the kernel: "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 " [1]. We assume that our implementation, since it is designed to deliver causal simulations, has a strong coverage over this directive (consider the TLM 2.0 temporal decoupling technique which often yields inaccurate simulations)

However, the feasibility of the introspection and code generation procedures, imposes certain limitations on SystemC's expressive capabilities. This is the main reason our kernel can not be considered to be compliant with the standard.

We also state that by assuming/enforcing the principle of one process per module and not allowing a module to execute another module's functions in its context (TLM 2.0 blocking transport interface), we hope to avoid causality errors caused by processes sharing system variables.

9.2 Process synchronization algorithm

We will begin our experimentations using a class of conservative synchronization algorithms originating from the work of **Chandy/Misra/Bryant (CMB)** [17, 18]. Listing 1 demonstrates how these algorithms deal with the fundamental dilemma presented in section 4, figure 4.

Algorithm 2 Process event loop, adopted from [19]

```
    while simulation is not over do
2:      Block until each incoming link queue contains at least one event
        remove event with the smallest time-stamp M from its queue.
4:      set clock = M
        process M
6:  end while
```

However, a naive realization of the algorithm leads to deadlock situations like the one depicted in figure 5. The queues placed along the red loop are empty, thus simulation has halted, even though there are pending events (across the blue loop).

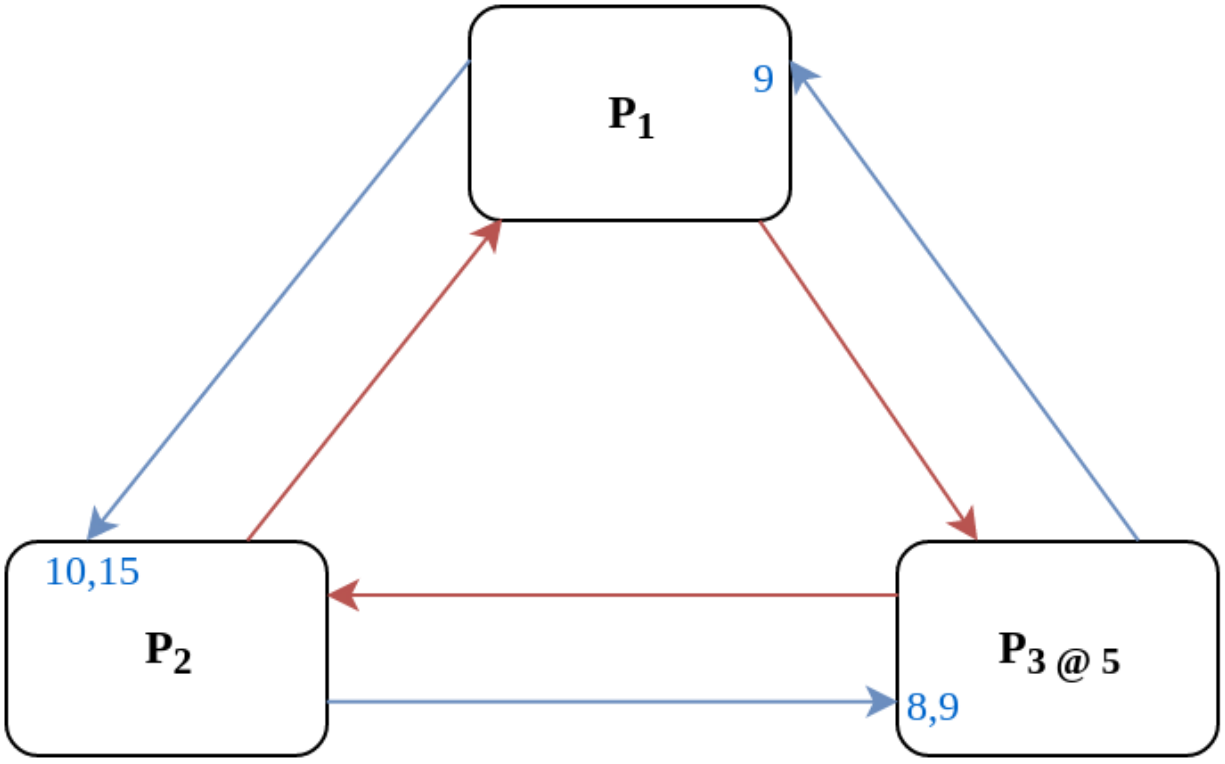

Figure 5: adopted from [19]

The deadlock avoidance mechanism that lies in the core of the CMB class of algorithms can be demonstrated with the following example: Let us assume that P_3 is at time 5. Furthermore, let us assume

that we have the **a priori** knowledge that P_3 has a minimum event processing time of 3 (simulated). We will call this knowledge **lookahead**. P_3 could create a **null event**, with no data value, but with a time-stamp $t(8)$ = clock(5) + lookahead(3) and place it on its outgoing links. A null event is still an event, so P_2 by processing it would advance its clock to 8. In the same fashion, let us assume that P_2 has a lookahead of 2 and upon processing P_3 null event, it will generate a null event for P_1 with time-stamp 10. Eventually P_1 can now safely process actual event with time-stamp 9, thus unfreezing the simulation.

The important points one must notice with this deadlock avoidance mechanism are that:

- Null events are created when a process updates its clock.
- Each process propagates null events on all of its outgoing links.
- This mechanism is mostly dependent to determine sufficiently large lookaheads.

9.3 Introspection and code generation

For the critical task of analyzing the model, identifying the processes and the links between them, we will follow ForSyDe SystemC's approach [20]. Using SystemC's well defined API for module hierarchy (e.g. `get_child_objects()`), along with the introduction of meta objects, the system's structure can be serialized at runtime, in the pre simulation phase of elaboration.

9.4 Hardware and Software tools

To ensure efficiency and code readability, we will use the explicit threading mechanisms that come with the latest standards of C++. The Intel Parallel Studio XE 2016 toolchain will be used for compilation, code analysis and optimization. We will initially use the Intel® Xeon Phi™ 5120D Coprocessor as the host platform for the simulation. The coprocessor is situated in a Intel® Xeon E5-2600M v3 server (named lovisa).

9.5 Evaluation Metrics

The first evaluation metric of the proposed kernel will be its strong scalability against the reference SystemC kernel. It will be determined by keeping the simulation's size constant and varying the number of processing elements. Furthermore, we will also measure weak scalability, by varying the number of processing elements and the simulation's size symmetrically, and trying to achieve constant time to simulation end.

The simulation's size can be easily related to the duration of the simulation (in simulated time). Another way of describing the simulation's size is through the conception of a formula involving the number of system processes, the number of links, the system's topology and the amount of events generated.

The accuracy of the simulation can be measured by the aggregate number of causality errors. The detection of causality errors must be facilitated in a per process level and the aggregation shall be performed at the end of the simulation. A concrete realization of the accuracy metric comes in the form of a counter each process increments whenever it executes an event with a time-stamp lower than its clock (the time-stamp of the last processed event).

10 Tasks and Time Scheduling

The majority of the remaining time has been partitioned into 3 x 3-4 week long iterations, where we will incrementally try to achieve our objectives. The tasks that comprise the first iteration are the following:

1. Setup software environment on the first experimentation server (Lovisa)
2. Manually compile the TLM 2.0 LT example of the SystemC installation, to a multithreaded C++ application, according to the CMB algorithm.

3. Document Task 1
4. Find a way to create parameterizable random networks within C++, that can be serialized in a form Matlab can analyze and visualize them.
5. Create "dummy" process functionality that resembles the behavior of common TLM actors (Instruction Set Simulators, Memory, Timers etc)
6. Perform a number of simulations of the random systems that correspond to the random networks.
7. Document Tasks 3-5.

The context of the next iterations will be decided based on the outcome of the first. To summarize:

Week(s)	End Date	Task Number	Milestone
5-7	02-18	Initial Investigation	Project Plan v1
7	02-19	1	
8	02-26	2	
9	03-02	3	Status Report 1
9	03-04	4	
10	03-11	5	
11	03-18	6	
12	03-23	7	Status Report 2
12-15	04-14	Second Iteration	Status Report 3
16-19	05-12	Third Iteration	Status Report 4
20-21		Reserve weeks	
22-26	06-30	Project closure	Final Report

11 References

[1] Open SystemC Initiative, *1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Std., 2012. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=6134617

[2] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, mar 1997. doi: 10.1109/5.558710. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=558710

[3] A. Jantsch and I. Sander, "Models of computation in the design process," 2005. [Online]. Available: http://people.kth.se/~ingo/Papers/IEE2005-Book.pdf

[4] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems

[5] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*, 2nd ed. Boston, MA: Springer US, 2010, no. 1. ISBN 978-0-387-69957-8. [Online]. Available: http://link.springer.com/10.1007/978-0-387-69958-5

[6] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES/ISSS '10*. New York, USA: ACM Press, 2010. doi: 10.1145/1878961.1879005. ISBN 9781605589053 p. 241. [Online]. Available: http://dl.acm.org/citation.cfm?id=1878961.1879005

[7] W. Chen, *Out-of-order Parallel Discrete Event Simulation for Electronic System-level Design*. Cham: Springer International Publishing, 2015. ISBN 978-3-319-08752-8. [Online]. Available: http://link.springer.com/10.1007/978-3-319-08753-5

[8] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, mar 2010. doi: 10.1109/DATE.2010.5457136. ISBN 978-3-9810801-6-2. ISSN 1530-1591 pp. 606–609. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5457136

[9] R. Domer, W. Chen, X. Han, and A. Gerstlauer, "Multi-core parallel simulation of System-level Description Languages," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*. IEEE, jan 2011. doi: 10.1109/ASPDAC.2011.5722205. ISBN 978-1-4244-7515-5 pp. 311–316. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5722205

[10] W. Chen, X. Han, C.-w. Chang, and R. Doemer, "Advances in Parallel Discrete Event Simulation for Electronic System-Level Design," *IEEE Design & Test of Computers*, vol. 30, no. October 2012, pp. 1–1, feb 2012. doi: 10.1109/MDT.2012.2226015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6338368

[11] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. New Jersey: IEEE Conference Publications, 2013. doi: 10.7873/DATE.2013.017. ISBN 9781467350716 pp. 9–14. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6513463

[12] T. Grötker, S. Liao, M. Grant, and S. Swan, *System Design with SystemC*. Boston: Kluwer Academic Publishers, 2002. ISBN 1-4020-7072-1. [Online]. Available: http://link.springer.com/10.1007/b116588

[13] Open SystemC Initiative, *OSCI TLM-2.0 language reference manual*, OSCI Std., 2009. [Online]. Available: http://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf

[14] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, oct 1990. doi: 10.1145/84537.84545. [Online]. Available: http://portal.acm.org/citation.cfm?doid=84537.84545

[15] ——, "Parallel and Distributed Simulation," in *Proceedings of the 2015 Winter Simulation Conference*, 2015. ISBN 978-1-4673-9743-8/15. [Online]. Available: http://www.informs-sim.org/wsc15papers/004.pdf

[16] A. Anane and E. M. Aboulhamid, "A Transaction-Based Environment for System Modeling and Parallel Simulation," *International Journal of Parallel Programming*, vol. 43, no. 1, pp. 24–58, feb 2015. doi: 10.1007/s10766-013-0303-4. [Online]. Available: http://link.springer.com/10.1007/s10766-013-0303-4

[17] R. E. Bryant, "Simulation of packet communication architecture computer systems," Cambridge, MA, USA, Tech. Rep., 1977. [Online]. Available: http://dl.acm.org/citation.cfm?id=889797

[18] K. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, sep 1979. doi: 10.1109/TSE.1979.230182. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702653

[19] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 978-0-471-18383-9

[20] S. Hosein, A. Niaki, M. K. Jakobsen, T. Sulonen, I. Sander, and D.-d. Oy, "Formal Heterogeneous System Modeling with SystemC," *Forum on Specification and Design Languages (FDL). IEEE*, pp. 160–167, 2012. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A586216&dswid=-7079