Parallel Simulation of SystemC Loosely-Timed Transaction Level Models

Master of Science Thesis

December 1, 2016

Author:	Konstantinos Sotiropoulos
Supervisor:	Björn Runåker (Intel Sweden AB)
Examiner:	Prof. Ingo Sander (KTH)
Academic adviser:	PhD student George Ungureanu (KTH)

KTH Royal Institute of Technology
School of Information and Communication Technology
Department of Electronic Systems
Stockholm, Sweden

Abstract

Parallelizing the development cycles of hardware and software is becoming the industry's norm for reducing electronic devices time to market. In the absence of hardware, software development is based on a virtual platform; a fully functional software model of a system under development, able to execute unmodified code.

A Transaction Level Model, expressed with the SystemC TLM 2.0 language, is one of the many possible ways for constructing a virtual platform. Under SystemC's simulation engine, hardware and software is being co-simulated. However, the sequential nature of the reference implementation of the SystemC's simulation kernel, is a limiting factor. Poor simulation performance often constraints the scope and depth of the design decisions that can be evaluated.

It is the main objective of this thesis' project to demonstrate the feasibility of parallelizing the co-simulation of hardware and software using Transaction Level Models, outside SystemC's reference simulation environment. The major obstacle identified is the preservation of causal relations between simulation events. The solution is obtained by using the process synchronization mechanism known as the Chandy/Misra/Bryantt algorithm.

To demonstrate our approach and evaluate under which conditions a speedup can be achieved, we use the model of a cache-coherent, symmetric multiprocessor executing a synthetic application. Two versions of the model are used for the comparison; the parallel version, based on the Message Passing Interface 3.0, which incorporates the synchronization algorithm and an equivalent sequential model based on SystemC TLM 2.0. Our results indicate that by adjusting the parameters of the synthetic application, a certain threshold is reached, above which a significant speedup against the sequential SystemC simulation is observed. Although performed manualy, the transformation of a SystemC TLM 2.0 model into a parallel MPI application is deemed feasible.

Keywords: parallel discrete event simulation, conservative synchronization algorithms, transaction level models, SystemC TLM 2.0

Acknowledgement

My Master's Thesis project was sponsored by Intel Sweden AB and was supervised by KTH's ICT department. Most of the work was carried out in Intel's offices in Kista, where I was kindly provided with all the necessary experimentation infastructure.

Björn Runåker was the project's supervisor from the company's side. I would like to thank you Björn for placing your trust in me for carrying out this challenging task. Furthermore, I would also like to thank Magnus Karlsson for his valuable feedback.

Professor Ingo Sander and PhD student George Ungureanu were the examiner and academic advisor from the university's side. I blame you for my intellectual Odyssey in the vast ocean of mathematical abstractions. I am now a sailor, on course for an Ithaka I may never reach. And I am most grateful for this beautiful journey. May our ForSyDe come true: the day when the conceptual wall between software and hardware collapses. *Let there be computation.*

Mother and father you shall be acknowledged, I owe my existence to you. Maria, I want to express my gratitude for your tolerance and support. Finally, Spandan, my comrade, you must always remember the price of intellect. Social responsibillity and chronic insomnia.

As you set out for Ithaka
hope the voyage is a long one,
full of adventure, full of discovery.

But do not hurry the journey at all.
Better if it lasts for years,
so you are old by the time you reach the island,
wealthy with all you have gained on the way,
not expecting Ithaka to make you rich.

Ithaka gave you the marvelous journey.
Without her you would not have set out.
She has nothing left to give you now.

And if you find her poor, Ithaka won't have fooled you.
Wise as you will have become, so full of experience,
you will have understood by then what these Ithakas mean.

Konstantinos Kavafis, Ithaka

Contents

List of Acronyms and Abbreviations

ASIC:	Application Specific Integrated Circuit
DE:	Discrete Event
DES:	Discrete Event Simulator/Simulation
DMI:	Direct Memory Interface
ES:	Electronic System
ESLD:	Electronic System-Level Design
FPGA:	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HPC:	High Performance Computing
IC	Integrated Circuit
IP	Intellectual Property
MoC:	Model of Computation
MPI	Message Passing Interface
MPSoC:	Multiprocessor System on Chip
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

List of Figures

1 Introduction

Section 1.1, provides an insight to the pragmatics of the project; without disclosing any commercially sensitive information, the reader is exposed to the use case, which became the raison d'être of this project. The problem definition is then presented in Section [BROKEN LINK: nil]. Section [BROKEN LINK: nil] presents the hypothesis; an optimistic assumption that motivated this work. Section 1.2 attempts to provide a general answer to the cui bono question. For a specific answer, the reader is encouraged to jump to section 6.4. Following the classification presented in [1], Section [BROKEN LINK: nil] describes the research methodology applied. Section [BROKEN LINK: nil] and 1.3 clarify the software engineering extend; what artifacts need to be constructed, in order to address the problem statement. A synopsys of this document can be found in 1.4

1.1 Overview

This project follows the work of Björn Runåker [2] on his effort to parallelize the simulation of the next generation (5G) of radio base stations. Telecom radio base stations are indeed a very heterogeneous system. To say the least, a virtual platform describing the system consists of a Network Processing Unit (NPU), Field Programmable Gate Array (FPGA) logic and a group of Digital Signal Procesors (DSP). For a more pictorial exposition of the situation the reader is encouraged to refer to the work of Björn.

The approach followed was defined as "coarse-grained"; parallelism is achieved through multiple instations of SystemC's simulation engine, one per major component. However, a question is left open; the feasibilty and merits of a "fine-grained" treatment, where parallelsim is achieved within a single instance of the simulation engine.

1.2 Purpose

An increasing amount of an Electronic System's (ES) expected use value is becoming software based. Companies which neglect this fact face catastrophic results. A well identified narrative, for example in [3], is how Nokia was marginalized in the "smartphone" market, despite possesing the technological know-how for producing superior hardware.

If an ES company is to withstand the economical pressure a competitive market introduces, the need for performing software and hardware development in parallel is imperative. Established ways of designing ESs, that delay software development until hardware is available, are therefore obsolete. The de facto standard of dealing with this situation has become the development of virtual platforms. It is obvious, that if a virtual platform is to be used for software development, it must be able to complete execution in the same order of magnitute as the actual hardware. Poor simulation performance often constraints the scope and depth of the design decisions that can be evaluated.

1.3 Delimitations

The following list demonstrates a number of artifacts that are not to be expected from this work, mainly due to their implementation complexity, given the limited time scope of a thesis project. However, one must keep in mind that the term "implementation complexity" often conceals the more fundamental question of feasibility.

- A modified version of the reference SystemC simulation kernel, capable of orchestrating a parallel simulation.

- A compiler for translating SystemC TLM 2.0 models into parallel applications. In fact, the previous statement should be generalized, for the shake of brevity: this thesis will not produce any sort of tool or utility.
- Any form of quantitative comparison between the proposed and existing attempts to parallelize SystemC TLM 2.0 simulations.

1.4 Structure of this thesis

2 Background

This chapter wishes to inform the reader about the theoretical constituents of this project. 2.1 presents the outermost context; that is the engineering discipline of **Electronic System-Level Design (ESLD)** and how SystemC TLM 2.0 fits into the whole picture. Section 2.2 hopes to help the reader understand why **Electronic System-Level Design Language** (ESLDL) models can be executed. In Section 2.3, SystemC's simulation engine is presented. This section is complemented by the code example found in Appendix A. Before proceeding, the reader is adviced to abandon momentarily any preconceptions about design, system, model, computation, time, concurrency and causality.

2.1 Electronic System-Level Design

Section 2.1.1 defines the fundamental concepts of design, system, model and simulation. In Sections 2.1.2 to 2.1.4, using Gajski and Kuhn's Y-Chart, the concept of a Transaction-Level Model is determined, as an instance in the engineering practice of Electronic System-Level Design (ESLD). Section 2.1.5 a rudimentary look on SystemC's role in ESLD.

2.1.1 The Design Process

We define the process of **designing** as the engineering art of incarnating a desired functionality into a perceivable, thus concrete, artifact. An engineering artifact is predominantly referred to as a **system**, to emphasize the fact that it can be viewed as a structured collection of components and that its behavior is a product of the interaction among its components.

Conceptually, designing implies a movement from abstract to concrete, fueled by the engineer's **design decisions**, incrementally adding implementation details. This movement is also known as the **design flow** and can be facilitated by the creation of an arbitrary number of intermediate artifacts called models. A **model** is thus an abstract representation of the final artifact in some form of a language. The design flow can be now semi-formally defined as a process of model refinement, with the ultimate model being the final artifact itself. We use the term semi-formal to describe the process of model refinement, because to the best of our knowledge, such model semantics and algebras that would establish formal transformation rules and equivalence relations are far from complete [4].

A desired property of a model is executability that is its ability to demonstrate portions of the final artifact's desired functionality in a controlled environment. An **executable model**, allows the engineer to form hypotheses, conduct experiments on the model and finally evaluate design decisions. It is now evident that executable models can firmly associate the design process with the scientific method. The execution of a model is also known as **simulation** [5].

2.1.2 Electronic Systems Design

An Electronic System (ES) provides a desired functionality, by manipulating the flow of electrons. Electronic systems are omnipotent in every aspect of human activity; most devices are either electronic systems or have an embedded electronic system for their cybernisis.

The prominent way for visualizing the ES design/abstraction space is by means of the Y-Chart. The concept was first presented in 1983 [6] and has been constantly evolving to capture and steer industry practices. Figure 1 presents the form of the Y-Chart found in [4].

The Y-Chart quantizes the design space into four levels of abstraction; system, processor, logic and circuit, represented as the four concentric circles. For each abstraction level, one can use different

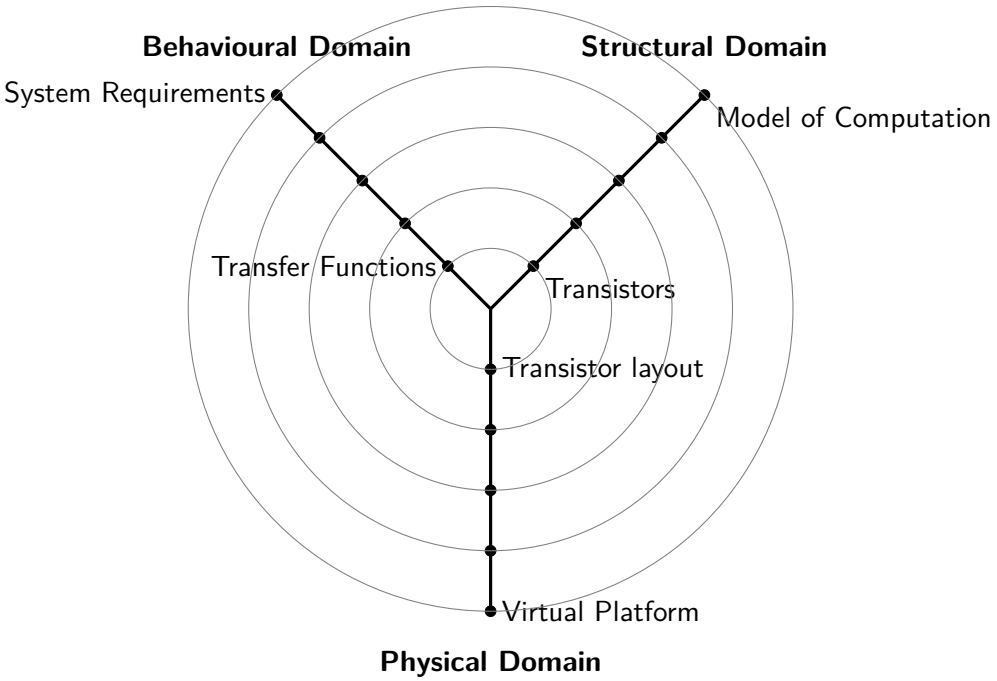

Figure 1: Gajski-Kuhn Y-chart

ways for describing the system: behavioral, structural and physical. These are represented as the three axises, hence the name Y-Chart. Models can now be identified as points in this design space.

A typical design flow for an Integrated Circuit (IC) begins with a high-level behavioral model capturing the system's specifications and proceeds non-monotonically to a lower level structural representation, expressed as a netlist of, still abstract, components. From there, Electronic Design Automation (EDA) tools will pick up the the task of reducing the abstraction of a structural model by translating the netlist of abstract components to a netlist of standard cells. The nature of the standard cells is determined by the IC's fabrication technology (FPGA, gate-array or standard-cell ASIC). Physical dimensionality is added by place and route algorithms, part of an EDA framework, signifying the exit from the design space, represented in the Y-Chart by the the "lowest" point of the physical axis.

The adjective non-monotonic is used to describe the design flow, because as a movement in the abstraction space, it is iterative: design $\rightarrow$ test/verify $\rightarrow$ redesign. This cyclic nature of the design flow is implied by the errors the human factor introduces, under the lack of formal model transformation methodologies in the upper abstraction levels. The term **synthesis** is also introduced to describe a variety of monotonic movements in the design space: from a behavioral to a less-equally abstract structural model, from a structural to a less-equally abstract physical model, or for movement to less abstract models on the same axis. Synthesis is distinguished from the general case of the design flow, in order to disregard the testing and verification procedures. Therefore, the term synthesis may indicate the presence, or the desire of having, an automated design flow. Low-level synthesis is a reality modern EDA tools achieve, while high-level synthesis is still a utopia modern tools are converging to.

2.1.3 System-Level Design

To meet the increasing demand for functionality, ES complexity, as expressed by their heterogeneity and their size, is increasing. Terms like Systems on Chip (SoC) and Multi Processor SoC (MPSoC), used for characterizing modern ES, indicate this trend. With abstraction being the key mental ability for managing complexity, the initiation of the design flow has been pushed to higher abstraction levels. In the Y-Chart the most abstract level, depicted as the outer circle, is the system level. At this level the distinction between hardware and software is a mere design choice thus **co-simulation of hardware and software** is one of the main objectives. Thereby the term **system-level design** is used to describe design activity at this level.

2.1.4 Transaction-Level Model

A **Transaction-Level Model** (TLM) can now be defined as the point in the Y-Chart where the physical axis meets the system abstraction level. As mentioned in the previous unit, a TLM can be thought of as a **Virtual Platform** (VP), where an application can be mapped [7]. Another way of perceiving the relationship between these three terms (TLM, VP and application) is to say the following: An application "animates" the virtual platform by making its components communicate through transactions. A TLM It is a fully functional software model of a complete system that facilitates **co-simulation of hardware and software**.

There are three pragmatic reasons that stimulate the development of a transaction level model. At first, as already mentioned, software engineers must be equipped with a virtual platform they can use for **software development**, early on in the design flow, without needing to wait for the actual silicon to arrive. Secondly, a TLM serves as a testbed for **architectural exploration** in order to tune the overall system architecture, with software in mind, prior to detailed design. Finally, a TLM can be a reference model for hardware **functional verification**, that is, a golden model to which an RTL implementation can be compared.

2.1.5 SystemC and TLM

One fundamental question, for completing the presentation of ESLD, remains; How can models be expressed on the system level? While maintaining the expressiveness of a Hardware Description Language (HDL), **SystemC** is meant to act as an **Electronic System Level Design Language** (ESLDL). It is implemented as a C++ class library, thus its main concern is to provide the designer with executable rather than synthesizable models. The language is maintained and promoted by Accellera (former Open SystemC Initiative OSCI) and has been standardized (IEEE 1666-2011 [8]). A major part of SystemC is the TLM 2.0 library, which is exactly meant for expressing TLMs. Despite introducing different language constructs, TLM 2.0 is still a part of SystemC because it depends on the same simulation engine. TLM 2.0 has been standardized seperately in [9].

2.2 The Discrete Event Model of Computation

With Section 2.2.1 the reader will be able to understand why a linguistic artifact, such as a model, can be "animated". In Sections 2.2.2 we present the **Discrete Event Model of Computation** (DE MoC). As with any MoC, the section presents what constitutes a component and what actions it can perform. Sections 2.2.3 and 2.2.4 define the concepts of causality, concurrency, time and determinism in the theoretical framework developed in the previous section.

2.2.1 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). The process of resolving the semantics of a linguistic artifact is called **computation**. Two approaches to semantics have evolved: denotational and operational. **Operational semantics**, which dates back to Turing machines, give the meaning of a language in terms of actions taken by some abstract machine. The word "machine" indicates a system that can be set in "motion" through "space" and time.

With operational semantics it is implied that a language can not determine computation byitself [10]. Computation is an epiphenomenon of the "motion" of the underlying abstract machine, just like time indication in a mechanical watch is a byproduct of gear motion. Consider the language of regular expressions. A linguistic artifact in this language describes a pattern that is either matched or not by a string of symbols. A Finite State Machine (FSM) is the underlying abstract machine. Computation is a byproduct of the FSM changing states; was the final state an accepting state or not. The rules that describe an abstract machine constitute a **Model of Computation (MoC)** [11].

All of the above painstaking narrative has been formed to reach the following conclusion: The dominant MoC related to an ESLDL is called the **Discrete Event (DE)** MoC, and it is the presence of the DE MoC that makes an ESLDL model executable.

2.2.2 Discrete Event Model of Computation

The components of a DE MoC are called **processes**. Processes introduce a spatial decompostion of a system; The system is mathematically represented as a set of variables $\mathbb{V}$, and every process is related to a subset of $\mathbb{V}$. The **system's state** is a mapping from $\mathbb{V}$ to a value domain $\mathbb{U}$. The system changes states in a discrete fashion; the set $\mathbb{A}$ of all possible system states can be enumerated by natural numbers ($\|\mathbb{A}\| = \aleph_0$).

A process can now be defined as a set of **events** $P_i \subseteq \mathbb{E}$ where $i \in \mathbb{N}$. $\mathbb{E}$ is a universal set on which processes P_i define a partition: $\bigcup_{i=1}^{n} P_i = \mathbb{E}$ and $P_i \cap P_j = \emptyset$ where $i, j, n \in \mathbb{N}$ and n is the number of processes. An event denotes a system state change; from the system's perspective, it can be regarding as a mapping $\mathbb{A} \to \mathbb{A}$.

$\mathbb{E}$ is a partially ordered set under the relationship **"happens before"**, denoted by the symbol $\sqsubset$ [12]. The binary relationship $\sqsubset$, apart from being antisymmetric and transitive, is irreflexive; an event can not "happen before" itself, it is counter intuitive to expect this when modeling physical systems.

On a process two actions are performed: communication and execution. Both of these can be defined as functions $\mathbb{E} \to \mathbb{E}$. **Execution** $f : P_i \to P_i$ is the processing of events (hence the name process to describe the entity that performs this action). In simpler terms, execution "consumes" an event, may change the system's state and may "produce" an event that needs to be communicated. Execution has the following property: $e_1 \sqsubset f(e_1)$ where $e_1 \in P_i$. In other words, the sets P_i are totally ordered under the $\sqsubset$ relationship. **Communication** $g : P_i \to P_j$ is the exchange of events.

In simpler terms, communication maps an event from one process to an event in another process. Communication has a similar property: $e_1 \sqsubset f(e_1)$ where $e_1 \in P_i$ and $f(e_1) \in P_j$.

2.2.3 Causality and Concurrency

The binary relationship **"causaly affects"**, denoted by the symbol $\propto$, is introduced. Causality, as a philosophical assumption about the behaviour of the system, can now be interpreted by the following statement: for any two events $e_1, e_2 \in \mathbb{E}$ it is true that $e_1 \propto e_2 \implies e_1 \sqsubset e_2$. Two events $e_1, e_2 \in \mathbb{E}$ are **concurrent** if neither $e_1 \sqsubset e_2$ nor $e_2 \sqsubset e_1$ holds. It follows, that concurrent events are not causaly related.

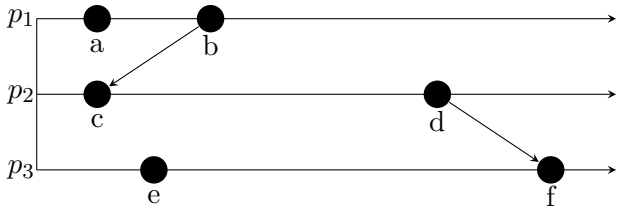

Figure 2: DE space-time decomposition

Figure 2 provides a visual understanding of a DE system, as a space-time diagram. A discrete perception of space is obtained by process decomposition (y-axis), while the perception of time (x-axis) is obtained by process actions. The horizontal arrows indicate process execution, while non-horizontal arrows indicate process communication. Events are represented as points in this plane. The execution and communication properties are denoted by placing the input event on the start of the arrow and the output event at its tip [1].

To move forward in time, one must follow a **chain** of ordered, under the $\sqsubset$ relationship, events. One such chain is the sequence a, b, c, d, f. Event a may causaly affect f. Events d, e are concurrent: there is no chain that contains both. Event d cannot causaly affet e and vice versa. The time axis is not resolved; a clocking mechanism for relating an event with a number, its timestamp, has not been defined. That is why the placement of events on the plane, for example events d, e is quite arbitrary, non-unique and maybe counter intuitive.

2.2.4 Time and Determinism

When implementing a DE MoC, one needs to differentiate between two notions of time: Simulated/logic time and real/wallclock time. **Real/Wallclock time** refers to the notion of time existing in the simulator; for example a x86 Time Stamp Counter (TSC), which counts the number of oscillation events since the reset event. **Simulated/logic time** is defined as a the notion of time in the DE. Since $\mathbb{E}$ is partially ordered and only the sets P_i are totally ordered, one is forced to reach the conclusion that the DE MoC instigates a **relativistic notion of time**. Simulated time may be different across processes, at any moment in real time. In other words, there is no global perception of logic time, that would allow, for example, the time axis in Figure 2 to be resolved/measured/quantized or, force a unique placement of events d, e in the plane.

Logic time modeling is deferred to the implementation of the DE abstract machine. It is highly depended on the nature of the undelying machine. Is it a **parallel** machine, that is a machine that preserves the spatial decomposition defined in the DE? Or is it a **sequential** machine, where space

[1] For execution, the reader has to imagine the presence of many intermediate arrows, between two subsequent events on the same horizontal arrow. The start is at the left event and the tip at the right.

dimensionality must be emulated. A realization of the DE abstract machine is called a **Discrete Event Simulator (DES)**.

If a DES can infer a total ordering of $\mathbb{E}$, somehow, then the simulation is said to be **deterministic**. A total ordering of $\mathbb{E}$ also infers a total ordering of the set $\mathbb{S}$: the system states encountered during simulation ($\mathbb{S} \subseteq \mathbb{A}$). Determinism is a very important reasoning facility, engineers seek from the simulation of the systems they construct, in order to provide any formal statement about the system's behavior. Physicists, especially those engaged with quantum mechanics, are more tolerant to non-determinism.

For amusement purposes only, the reader can regard her/his brain as a DES. How does the human brain handles the relativistic nature of time; it infers total orderings for the events of reality. Alas, human intuition is biased towards a deterministic understanding of the physical world. Intellect, though, is (hopefully) much more capable!

2.3 SystemC's Discrete Event Simulator

Section 2.3.1 demonstrates how SystemC realizes the concept of a process. This section is complemented by the code example found in Appendix A.

2.3.1 Coroutines

SystemC's distribution comes with a sequential realization of the DE MoC, referred to as the reference **SystemC simulation engine** [8]. It is a sequential implementation because the spatial decomposition of the system is emulated through **coroutines** (also known as co-operative multitasking). Co-routines in SystemC have been counterintuively named as `SC_METHOD`, `SC_THREAD` or `SC_CTHREAD`. A coroutine is neither a function nor a thread.

Processes, realized as coroutines[2], perform their actions (computation, communication), henceforth **run**, without interruption. At any moment in real time only a single process can be running. No other process can run until the running process has voluntarily **yielded**. Furthermore, a non-running process can not preempt or interrupt the running process.

A process can be declared sensitive to a number of events (static sensitivity). Moreover, a process can declare itself sensitive to events (dynamic sensitivity). All of the events the process is sensitive to, form its **sensitivity list**. A yielded process is awaiting for events in its sensitivity list to to be triggered.

Before yielding, a process saves its context and registers its identity in a global structure of coroutine handlers called the **waiting list**. Along comes the question: to whom does a yielding process pass the baton [3] of control flow?

2.3.2 The kernel

The **kernel** is the simulation's director [5], the maestro of a well orchestrated simulation music. Processes yield to the kernel, a coroutine himself. In the presence of an ill-behaved never yielding process, the kernel is powerless [4].

The kernel is responsible for many things[5]:

- It sorts the **global event queue** according to timestamp order.
- It is from his perspective that a non-relativistic notion of logic time is formed: it maintains a **clock** that advances according to the timestamp of the event last triggered.
- When the list of **runnable** coroutines has been depleted, it is his duty to trigger the next, according to timestamp order, event.
- When triggering an event, it must identify which processes can be moved from the waiting to the runnable list. The decision is based on a process sensitivity list.

[2]The exact library that realizes co-routines in C++ is determined during the compilation of the SystemC distribution. In GNU/Linux, SystemC version 2.3.1 supports QuickThreads and Posix Threads. However, it is highly propable that future revisions of the C++ standard will include **resumable functions**, a concept semantically equivalent to coroutines.

[3]The short stick being passed on in relay races.

[4]This is exactly the most important problem faced by early operating systems (16-bit era). Their cooperative nature could not discipline poorly designed applications.

[5]Please note that many terms are forward-declared and defined either further down in the description or in upcoming sections.

- It is responsible for **context switching** between the running and a runnable process. The selection of the running process from the list of runnable processes is arbitrary. Under certain conditions, this is a backdoor for non-determinism. An example of such a situation can be found in Appendix B.
- If there are no events in the global event queue and the list of runnable processes is empty, it must **terminate** the simulation.

A spectre is haunting the previous description of the kernel: how is logic time modeled?

2.3.3 Modeling Time

Logic time is represented as a vector [6] $(t, n) \in \mathbb{N}^2$. Every event is associated with the moment in logic time it occured. In other words, every event has a **timestamp**. Ordering of events comes as a lexicographical comparison between timestamps. Two events e_1, e_2 associated with the timestamps $(t_1, n_1), (t_2, n_2)$ are said to be **simultaneous** if $t_1 = t_2$. If both $t_1 = t_2$ and $n_1 = n_2$ they are **strongly simultaneous**.

The first co-ordinate of a logic time vector is meant for modeling real time. **Modeled real time values** are used as timing annotations the designer injects into the system in order to describe the duration of communication and execution in the physical system. To avoid quantization errors and the non-uniform distribution of floating point values, time is internally represented as an integral multiple of an SI unit referred to as the time resolution. The integral multiplier is limited by the underlying machine's capabilities: in a 64-bit architecture its maximum value is $2^{64} - 1$. The minimum time resolution SystemC can provide is that of a femtosecond (10^{-15} seconds).

To assist in the construction of modeled real time values, SystemC provides the class `sc_time`. `sc_time`'s constructor takes two arguments: (`double`, `SC_TIME`) [7]. The designer needs to be very careful when providing time annotations: modeled real time is internally represented as an integral value, despite `sc_time`'s constructor having a floating point argument. The mistake of using a value of `sc_time(0.5, SC_FS)` can only be detected during **run-time**. The same applies for a value of `sc_time(1, SC_SEC)` with a time resolution of 1 `SC_FS`.

2.3.4 Events

Events in SystemC are realized as instances of the class `sc_event`. A process explicitly "makes" an event occur by calling either of these variations of the `sc_event.notify` method:

- `notify()`: Immediate occurence.
- `notify(SC_ZERO_TIME)`: Delayed occurence.
- `notify(sc_time t)`: Scheduled occurence.

Now that a metric for logic time has been established, we be more specific on a process yields. Yielding is explicitly stated by a calling:

- `wait()`
- `wait(sc_time)`
- `wait(sc_event)`

[6] This time modeling technique is referred to as **superdense time** in [5]. However, this is not consistent across literature, for example in [13].

[7] `SC_TIME` is an enumeration: `SC_SEC` for a second, `SC_MS` for a millisecond etc.

2.3.5 The Simulation Procedure

SystemC event loop, adopted from [14] [1]

run all processes order events scheduled events exist Simulation time progression order events and trigger the event with the smallest timestamp advance simulation time make all sensitive processes runnable runnable processes exist Delta cycle progression runnable processes exist run all runnable processes trigger all immediate notifications make all sensitive processes runnable trigger all delayed events make all sensitive processes runnable

2.4 Parallel Discrete Event Simulation

Units [BROKEN LINK: nil] and [BROKEN LINK: nil] introduce the concept of Parallel Discrete Event Simulation (PDES) and identify the fundamental causality hazards.

2.4.1 Prior Art

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 interleaved process execution to 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)**. After making the strategical decision that for improving a DE simulator's performance one must orchestrate parallel execution, the first tactical decision encountered is whether to keep a single simulated time perspective, or distribute it among processes.

For PDES implementations that enforce global simulation time, the term **Synchronous PDES** has been coined in [14]. In Synchronous PDES, parallel execution of processes is performed within a delta cycle. With respect to Alg 2.3.5, we can say that a Synchronous PDES parallelizes the execution of the innermost loop (line 4). However, as we will see in later sections, this approach will bare no fruits in the simulation of TLM Loosely Timed simulations, since delta cycles are never triggered [15]. Therefore, we switch our interest in **Out-of-Order PDES (OoO PDES)** [16]; allowing each process to have its own perception of simulated time, determined by the last event it received.

2.4.2 Causality Hazards

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 [17]. 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.

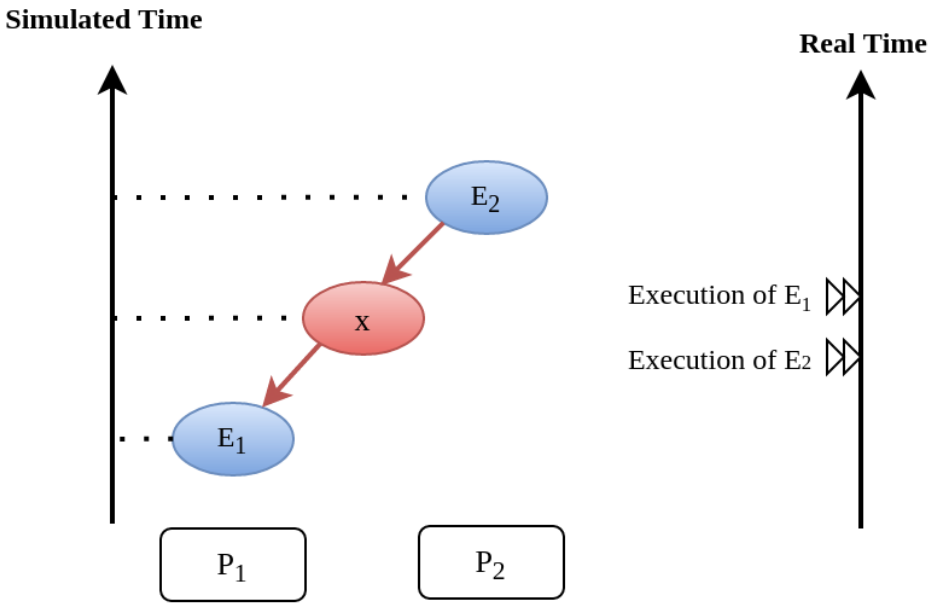

Figure 3: Causality error caused by the sharing of the system's state variable x by P_1 and P_2.

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 sake 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 [17]? 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 $t_2 < t_1$".

OoO PDES synchronization algorithms, with respect to how they deal with causality errors, have been classified into two categories: **conservative** and **optimistic** [18]. 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

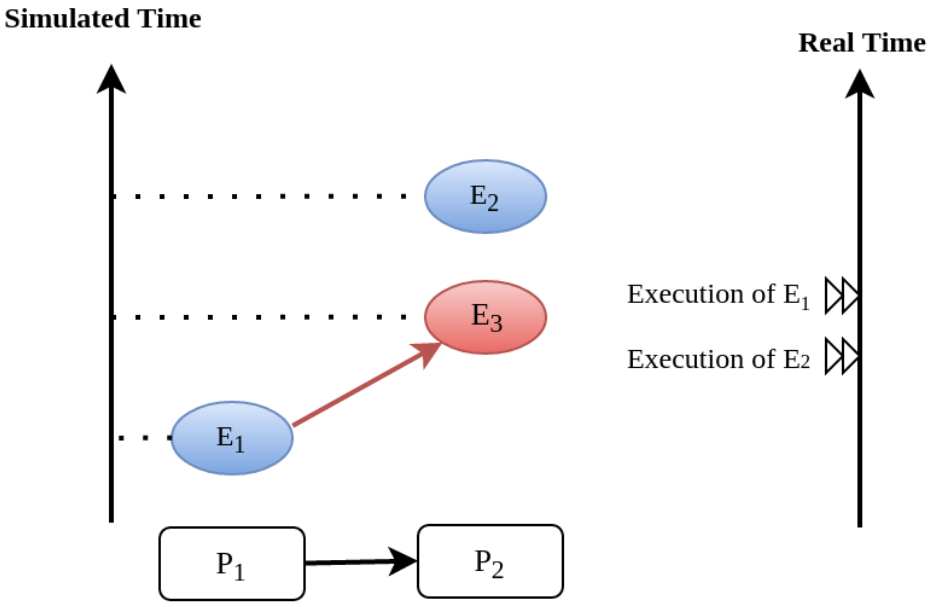

Figure 4: Causality error caused by the unsafe execution of event E_2 (adopted from [17]).

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 [17].

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 [19] .

3 Methodology

4 Out of Order PDES with MPI

The goal of this chapter is to present the process synchronization algorithm that will be applied and give their implementation using the MPI API.

In units 4.1 and [BROKEN LINK: nil] we present the conservative synchronization algorithms that will be evaluated. In unit 4.4 and 4.5 we present the semantics of the Message Passing Interface (MPI) communication primitives. In unit 4.6 we provide pseudo code for the realization of the CMB using the MPI communication primitives. In unit 4.8 we give an overview of prior art in the field of PDES in ESLD.

4.1 The Chandy/Misra/Bryant synchronization algorithm

The synchronization algorithm at the heart of the proposed OoO PDES is known as the **Chandy/Misra/Bryant (CMB)** [20] [21]. Historically, it has been the first of the family of conservative synchronization algorithms [17].

According to the algorithm, the physical system to be simulated must be modeled as a number of communicating sequential **processes**. The system's state, a set of variables, is distributed in a disjoint way, across the processes. Computation is reactive; it is sparked by an event and produces further events and **side-effects** (changes in a subset of the system's variables). Each process keeps its own perspective of simulated time through a **clock** variable. The value of the clock is equal to the timestamp of the last event selected for computation.

Based on the system's state segregation, a static determination of which processes are interdependent can be established. This is indicated by placing a **link** for each pair of dependent processes. From a process' perspective a link can be either **outgoing**, meaning that events are sent via the link, or **incoming** meaning that events are received through it. An incoming link must encapsulate a First-In-First-Out (FIFO) data structure for storing incoming events, in the order they are received.

The order by which events are received is **chronological**; non decreasing timestamp order. This system-wide property is maintained by making each process select for computation the event that has the smallest timestamp. A formal proof of how this local property **induces** a system-wide property can be found in [20] [21]. Chronological reception of events is a necessary, but not sufficient, condition for ensuring **causality**. The algorithm deals with the "is an event safe to execute" dilemma by forcing a process to **block** until each of its incoming links contains an event. All the above are demonstrated in Listing 2.3.5. The synchronization algorithm is realized as a process' main event loop.

Process event loop, without deadlock avoidance [1]

process clock $<$ some T **Block** until each incoming link FIFO contains at least one event Pop event M, with the **smallest** timestamp across all incoming links. Set process' **clock** = timestamp(M) **React** to event M **Communicate** resulting events over the appropriate links

4.2 Deadlock Avoidance

The naive realization of the process' event loop presented in Listing 2.3.5 leads to deadlock situations like the one depicted in Figure 5. The queues placed along the outer loop are empty, thus simulation has halted, even though there are pending events (across the queues of the inner loop). A global simulation moderator could easily detect deadlocks and allow the process, that has access to the event with the global minimum timestamp, to resume execution. The presence of a moderator, however, would violate the distributed nature of the simulation, thus increasing the implementation complexity of the simulation environment. Furthermore,

For the context of this thesis, a distributed mechanism is more favorable. What follows is a presentation of a distributed mechanism for overcoming these situations, referred to as the **null-event deadlock avoidance** [22].

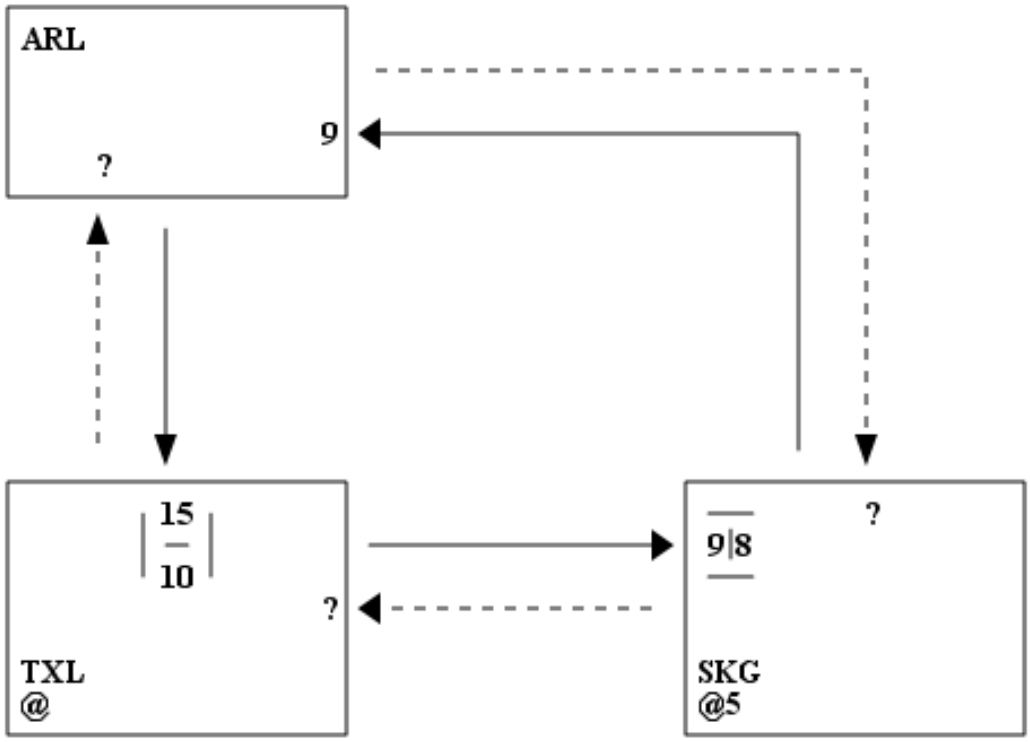

Figure 5: Deadlock scenario justifying the use of Null messages in the CMB

Figure demonstrates an air traffic simulation, where the airports (ARL, CDG and SKG) constitute the simulation processes. The events exchanged between the airports model flights, the time unit being arbitrary. At deadlock, every airport is at time 5.

Furthermore, it is assumed that there is an **a priori** knowledge conserning the flight time between airports. This knowledge is referred to as the **lookahead** and takes the form of a function $lookahead : (PxP) \rightarrow time$ For example, by selecting the distance between every airport to be 3 time units, one can deduce the following: since SKG is at 5 then ARL or CDG should not expect any event from SKG before 8.

To communicate this fact, SKG could create a special kind of event, a **null event**, with no data value, but with a timestamp 8 (clock+lookahead) and place it on its outgoing links. A null event is still an event, so CDG would acknowledge it during the selection phase, thus being able to receive the flight from ARL. CDG now sits at 5 and in the same fashion it could broadcast a null event with timestamp 8. It is evident that the deadlock has been solved, at the expense of flooding the communication links with null events.

Process event loop, with deadlock avoidance [1]

process clock < some T **Block** until each incoming link FIFO contains at least one event Remove event M with the smallest timestamp from its FIFO. Set process' clock = timestamp(M) **React** to

event M **Communicate** either a null or meaningful event to each outgoing link with timestamp = clock + lookahead

4.3 Criticism

The modified, for deadlock avoidance, algorithm is described in listing 4.2. The important points one must notice with this deadlock avoidance mechanism are that:

- Null events are created when a process updates its clock, that is upon processing an event.
- Each process propagates null events on all of its outgoing links.
- The efficiency of this mechanism is highly dependent on the designer's ability to determine sufficiently large lookaheads.
- The lookahead must be a function

4.4 Semantics of point-to-point Communication in MPI

There is a problem here: There are two sections. Semantics of Nonblocking and Blocking communications in the MPI manual

The framework chosen for implementing the PDES is the **Message Passing Interface** 3.0 (MPI). Events are modeled as structured messages, while event diffusion/communication as message passing. MPI is a message passing library interface specification, standardized and maintained by the Message Passing Interface Forum [23]. It is currently available for C/C++, FORTRAN and Java from multiple vendors (Intel, IBM, OpenMPI) [23]. MPI addresses primarily the message passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process [24].

The basic communication primitives are the functions `MPI_Send(...)` and `MPI_Recv(...)`. Their arguments specify, among others things, a data buffer and the peer process' or processes' unique id assigned by the MPI runtime. By default, message reception is blocking, while message transmission may or may not block. One can think of message transfer as consisting of the following three phases

1. Data is pulled out of the send buffer and a message is assembled
2. A message is transferred from sender to receiver
3. Data is pulled from the incoming message and disassembled into the receive buffer

Order: Messages are non-overtaking. If a sender sends two messages in succession to the same destination, and both match the same receive (a call to `MPI_Recv`), then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives and also guarantees that message passing code is deterministic.

Fairness: MPI makes no guarantee of fairness in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. It is the programmer's responsibility to prevent starvation in such situations.

COMMENT: Why did you choose MPI?

4.5 MPI Communication Modes

The MPI API contains a number of variants, or **modes**, for the basic communication primitives. They are distinguished by a single letter prefix (e.g. `MPI_Isend(...)`, `MPI_Irecv(...)`). As dictated by the MPI version 3.0, the following communication modes are supported [24]:

No-prefix for standard mode: `MPI_Send(...)` In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete, blocking the transmitting process, until a matching receive has been posted, and the data has been moved to the receiver.

B for buffered mode: `MPI_Bsend(...)` A buffered mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is local, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signaled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values and sends them to a consumer. Assume that the producer produces new values faster than the consumer can consume them. If buffered sends are used, then a buffer overflow will result. Additional synchronization has to be added to the program so as to prevent this from occurring.

S for synchronous mode: `MPI_Ssend(...)` A send that uses the synchronous mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes **rendezvous** at the communication point.

R for ready mode: `MPI_Rsend(...)` A send that uses the ready communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. Ready sends are an optimization when it can be guaranteed that a matching receive has already been posted at the destination. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead.

Maybe you should consider non-blocking communication not as a **mode**.

I for non-blocking mode: `MPI_Isend(...)`, `MPI_Ibsend(...)`, `MPI_Issend(...)` and {MPI\Irecv`(...)` Non-blocking message passing calls return control immediately (hence the prefix I), but it is the user's responsibility to ensure that communication is complete, before modifying/using the content of the data buffer. It is a complementary communication mode that works en tandem with all the previous. The MPI API contains special functions for testing whether a communication

is complete, or even explicitly waiting until it is finished.

4.6 MPI Realization of CMB

Listing 4.6 is a pseudo code, sketching out the CMB process event loop, using MPI's communication primitives. CMB Process event loop in MPI [2]

process clock < some T post a MPI_Irecv on each incoming peer process post a MPI_Wait: block until every receive has been completed save each message received in a separate, per incoming link, FIFO. identify message M with the smallest time-stamp set clock = time-stamp(M) process message M post a MPI_Issend to each outgoing link L with time-stamp = clock + Lookahead(clock,L,...)

Applications have specific communication patterns

Also provides information about the application's communication behavior to the MPI implementation.

Topology mapping One of the major features of MPI's topology interface is that it can easily be used to adapt the MPI proces layout to the underlying network and system topology.

non cartesian topologies

What is the implementation type of the event? Let us custom pack them in one 64 bit integer. Extract them by mapping.

Since you always send an event to your neighbors, either a meaningfull one or a null, why not broadcast?

4.7 Evaluation Metrics

The first evaluation metric of the proposed PDES implementation will be its performance against the reference SystemC kernel. It will be measured by experimentation on the project's use case.

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). Ideally, if the synchronization algorithms have been realized correctly, no causality errors should be detected.

COMMENT: This section will become more concrete when we start experimentation.

4.8 Existing PDES

The most important: RISC: Recoding infrastructure for SystemC [25].

Miscellaneous: SystemC-SMP [26] SpecC [27], although the latter is not meant for SystemC. $sc\backslash_{during}$ [28]

COMMENT: This section is incomplete that should not be incomplete in an Intermediate report. Are you reinventing the wheel? Did you try at least one of these tools?

5 Analysis

6 Conclusion and Future Work

The major contributions of this work can be found in Section 6.1. Section 6.2 provides a list of actions that the author believes that should have been performed This work is far from complete: The brave Theseus that would like to confront the minotaur can find Ariadne's thread in Section 6.3 Section 6.4 revisits, in a more specific way, the cui bono question answered in Section 1.2.

6.1 Contributions

The following are the main research contributions of this work:

- In Section 2.2 a different approach is adopted for presenting the DE MoC, when compared to the reference work in MoCs by the Ptolemy Project [8]. It is the fact that time modeling is not included in the description of the DE MoC itself; Time modeling is an implementation concern. For the abstract/mathematical description of the DE MoC, Lamport's "happens before" relationship [12] suffices in describing the important concepts emanating (e.g. causality, concurrency and determinism).

6.2 Limitations

- The theoretical description of the DE MoC in section 2.2 is far from complete. It lacks of a Turing completeness proof.
- Intel's Xeon Phi coprocessor was not used as an experimentation tool, despite this being specified as a primary objective in the project plan. Its Multiple Instruction Multiple Date (MIMD) architecture and its highly parametirized MPI implementation, makes it an ideal platform for performing the proposed OoO PDES simulation. However, we are able to report that SystemC 2.3.1 can be compiled with Intel's C++ compiler 16.0 for the Xeon Phi platform. Moreover, the compiled package was verified againt the accompanying test suite.
- Not establishing an open communication channel with the following two scientists/engineers/researchers: Professor Rainer Dömer [9] and Dr. Jakob Engblom [10] It is a researcher's ethical obligation towards society to take the initiative for disseminating his work. This work could be of some infinitesimal value towards the important, for the collective, work they do on ES design. Vice versa, their feedback would have greatly increased the quality of the work.

6.3 Future Work

Unfortunately the library of the or1ksim is not reentrant and thus does not allow multiple instances of the core simulator to be executed in one address space. Historically all data is stored in global variables.

In Section 1.3, the automatic compilation of a SystemC TLM 2.0 model into our proposed MPI implementation was indicated as a delimitation of this project. However, it is the next logical step in progressing this work, since it has been deemed feasible. Some general guidlines are:

[8] The Ptolemy Project, Center for Hybrid and Embedded Software Systems (CHESS), Department of Electrical Engineering and Computer Sciences, University of California at Berkeley: http://ptolemy.eecs.berkeley.edu/

[9] Professor Rainer Dömer works at the University of California Irvine, The Henry Samueli School of Engineering: http://www.cecs.uci.edu/~doemer/. His current project *Parallel SystemC Simulation on Many-Core computer architectures* is highly relevant to this thesis.

[10] Dr. Jakob Engblom works as a Product Management Engineer at Intel in Uppsala: https://www.linkedin.com/in/jakobengblom. His academic research and professional experience with virtual platforms would be a significant source of feedback.

- For the critical task of analyzing the model (identifying the processes and the links between them), ForSyDe SystemC's approach could be mimicked [29]. 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.

- After elaboration simulation should halt. The desirable outcome, propably in some XML format, was the serialization of the system's structure. The proposed compiler can now use this abstract representation in conjunction with a library of code skeletons to generate the desired MPI implementation.

Although not relevant to the thesis, during the implementation of the cache hierarchy, the author has identified the need for an open-source framework for designing, documenting, implementing and testing FSMs. TikZ-UML could serve as the front-end. It can express most of the UML 2.0 statechart defined concepts and produce a visual representation. Since the syntax follows a structural manner, a compiler for the following backends could be developed:

- NuSMV for model checking by expressing requirements as temporal logic expressions.

- Quantum Leaps can provide a well structured, easily maintained and tested C/C++ real-time implementation.

Furthermore, Emacs' Org mode could be used for housing the compilation procedure, by unifying the editing of all the above representations of the FSM. Emacs Org mode is more than a text editor: it is an ecosystem that enables the symbiosis of source code and document, in an unpresented way, that follows Donald Knuth concept of literate programming. It is an indispensable tool when reproducibility is a desirable feature [30].

6.4 Reflections

On May the 3rd 2016 the SystemC user community came together at Intel's headquarters in Munich, for a full-day workshop about the evolution of the various SystemC standards. The event was called SystemC Evolution Day 2016 [11] and was organized by Accelera, the organization responsible for advancing the language. Professor Rainer Dömer gave a highly influential presentation titled *"Seven Obstacles in the Way of Parallel SystemC Simulation"*, from where the following views can be induced:

- A formal understanding of the DE MoC is needed.

- The progression from sequential DES to PDES is of vital importance for the longevity of the language. As Professor Dömer humorously remarks: *"SystemC must embrace true parallelsim otherwise it will go down the same path as the dinosaurs"*

The fact that that this project's initiaton preceeds ($\sqsubset$) the event, can be regarded as an indication of proper alignment: this project is organically bound to the ongoing discussion about SystemC's new major revision.

[11] All presentations from the event are available at: [http://accellera.org/news/events/systemc-evolution-day-2016]

7 References

[1] A. Håkansson, "Portal of Research Methods and Methodologies for Research Projects and Degree Projects," *International Conference on Frontiers in Education: Computer Science and Computer Engineering*, vol. 13, pp. 67–73, 2013. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A677684&dswid=9928

[2] B. Runåker, "Distributed system simulation with host-based target offloading," Master's thesis, KTH Royal Institue of Technology. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?dswid=-9606&pid=diva2%3A807591&c=1&searchType=SIMPLE&language=en&query=run%C3%A5ker&af=%5B%5D&aq=%5B%5B%5D%5D&aq2=%5B%5B%5D%5D&aqe=%5B%5D&noOfRows=50&sortOrder=author_sort_asc&onlyFullText=false&sf=allo

[3] J. Surowiecki, "Where Nokia Went Wrong," New York, NY, USA, sep 2013. [Online]. Available: http://www.newyorker.com/business/currency/where-nokia-went-wrong

[4] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design*. Boston, MA: Springer US, 2009. ISBN 978-1-4419-0503-1. [Online]. Available: http://link.springer.com/10.1007/978-1-4419-0504-8

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

[6] Gajski and Kuhn, "Guest Editors' Introduction: New VLSI Tools," *Computer*, vol. 16, no. 12, pp. 11–14, dec 1983. doi: 10.1109/MC.1983.1654264. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1654264

[7] S. Rigo, R. Azevedo, and L. Santos, Eds., *Electronic System Level Design*. Dordrecht: Springer Netherlands, 2011. ISBN 978-1-4020-9939-7. [Online]. Available: http://link.springer.com/10.1007/978-1-4020-9940-3

[8] 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

[9] ——, *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

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

[11] 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

[12] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, 1978. doi: 10.1145/359545.359563. [Online]. Available: http://dl.acm.org.focus.lib.kth.se/citation.cfm?doid=359545.359563

[13] C. a. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, *Modeling time in computing*, 2010, vol. 42, no. 2. ISBN 978-3-642-32331-7. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1667062.1667063

[14] 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

[15] 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

[16] 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

[17] 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

[18] ——, "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

[19] 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

[20] 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

[21] 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

[22] 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

[23] citation needed, *citation needed*.

[24] Message Passing Interface Forum, *MPI: A Message Passing Interface Standard Version 3.0*. Knoxville, Tenessee: University of Tenessee, 2012. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[25] G. Liu, T. Schmidt, D. Rainer, G. Liu, T. Schmidt, and D. Rainer, "RISC Compiler and Simulator , Alpha Release V0 . 2 . 1 : Out-of-Order Parallel Simulatable SystemC Subset," Center for Embedded and Cyber-physical Systems University

of California, Irvine, Irvine CA USA, Tech. Rep. 949, 2015. [Online]. Available: http://www.cecs.uci.edu/{~}doemer/publications/CECS{_}TR{_}15{_}02.pdf

[26] 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

[27] 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

[28] 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

[29] 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

[30] E. Schulte and D. Davison, "Active documents with org-mode," *Computing in Science and Engineering*, vol. 13, no. 3, pp. 66–73, 2011. doi: 10.1109/MCSE.2011.41

Appendices

A Producer Consumer Example in SystemC

This example complements the presentation of the SysteC simulation engine in Section [BROKEN LINK: nil]. It is an example of a producer-consumer system. The producer communicates with the consumer via a fifo channel. Its primary purpose is to demystify the way primitive channels are implemented in SystemC, by revealing their event driven nature.

These are the interfaces the channel must be able to provide to the actors:

```
#include "systemc.h"
#include <iostream>
using namespace sc_core;

//****************
// FIFO Interfaces
//****************
// Fifo interface exposed to Producers
class fifo_write_if: virtual public sc_interface
{
public:
    // blocking write
    virtual void write(char) = 0;
    // number of free entried
    virtual int numFree() const = 0;
};
// Fifo interface exposed to Consumers
class fifo_read_if: virtual public sc_interface
{
public:
    // blocking read
    virtual char read() = 0;
    // number of available entries
    virtual int numAvailable() const = 0;
};
```

Following, is the interface of the fifo channel, which internally acts like a circular buffer.

```
//*************
// FIFO channel
//*************
class Fifo: public sc_prim_channel, public fifo_write_if, public fifo_read_if
{
protected:
    int   size;
    char *buf;
    int   free;
    int   ri; // read index
```

```
    int   wi; // write index
    int   numReadable, numRead, numWritten;
    // For notifying Producer and Consumer
    sc_event Ev_dataRead;
    sc_event Ev_dataWritten;
public:
    // Constructor
    explicit Fifo(int _size=16):
        sc_prim_channel(sc_gen_unique_name("thefifo"))
        {
            size = _size;
            buf = new char[_size];
            reset();
        }
    // Destructor
    ~Fifo(){ delete [] buf; }
    int  numAvailable() const { return numReadable - numRead; }
    int  numFree() const { return size - numReadable; }
    void reset() { free=size; ri=0; wi=0; }
    void write(char c);
    char read();
    void update();
};
```

Next we see how the channel realizes the blocking read and write interfaces. The `read` and `write` methods are executed during the **evaluation phase**. The co-routine that executes these methods will yield immediately if it reaches the `wait` statement. When an event is passed as an argument to the `wait` function, the co-routine's sensitivity is said to change dynamically. The `request_update` method (inherited from `sc_prim_channel`) is a kernel callback. It signals the kernel that during the **update phase** he should execute the channel's `update` method.

```
// Blocking write implementation
void Fifo::write(char c)
{
    if (numFree() == 0)
        wait( Ev_dataRead );
    numWritten++;
    buf[wi] = c;
    wi = (wi+1) % size; // Circular buffer
    free--;
    request_update();
}
// Blocking read implementation
char Fifo::read()
{
    if (numAvailable() == 0)
        wait( Ev_dataWritten );
    numRead++;
```

```
    char temp  = buf[ri];
    ri = (ri+1) % size; // Circular buffer
    free++;
    request_update();
    return temp;
}
```

Following, is the implementation of the `update` method, which is executed during the update phase by the kernel's. A yielded (**blocked**) co-routine might end up in the **runnable** set if it has declared its sensitivity to the event being notified.

```
// Update method called in the UPDATE phase of the simulation
void Fifo::update()
{
    if (numRead > 0)
        Ev_dataRead.notify(SC_ZERO_TIME);
    if (numWritten > 0)
        Ev_dataWritten.notify(SC_ZERO_TIME);
    numReadable = size - free;
    numRead = 0;
    numWritten = 0;
}
```

Next we see the implementation of the producer and consumer modules. The co-routine is declared sensitive (static sensitivity) to a clock's rising edge. The co-routine that represents these modules executes the `run` function. Since all co-routines are declared runnable at **elaboration**, they need to yield immediately after entering the function.

```
class Producer: public sc_module
{
public:
    sc_port<fifo_write_if> out;
    sc_in<bool> clock;
    void run()
        {
            while(1)
            {
                wait(); // wait for clock edge
                out->write(1);
                cout << "Produced at: " << sc_time_stamp() << endl;
            }
        }
    // Constructor
    SC_CTOR(Producer)
        {
            SC_THREAD(run);
            sensitive << clock.pos();
```

```
        }
};


class Consumer: public sc_module
{
public:
    sc_port<fifo_read_if> in;
    sc_in<bool> clock;
    void run()
        {
            while(1)
            {
                wait(); // wait for clock edge
                char temp = in->read();
                cout << "Consumed at: " << sc_time_stamp() << endl;
            }
        }
    SC_CTOR(Consumer)
        {
            SC_THREAD(run);
            sensitive << clock.pos();
        }

};
```

Finally, the modules are linked with the fifo and their clock, and simulation is started.

```
int sc_main(int argc, char *argv[])
{
    sc_clock clkFast("ClkFast", 1, SC_NS);
    sc_clock clkSlow("ClkSlow", 500, SC_NS);

    Fifo fifo1;

    Producer p1("p1");
    p1.out(fifo1);
    p1.clock(clkFast);

    Consumer c1("c1");
    c1.in(fifo1);
    c1.clock(clkSlow);

    sc_start(5000, SC_NS);

    return 0;
}
```

B Non-Determinism in SystemC

The following code example should in theory lead to non-deterministic behavior. It models a race condition. The system contains 3 processes which access a sharedVariable: 2 of them write it and 1 reads it. At every clock pulse, all 3 processes are made **runnable**. In practice however there is a repeatable pattern: processes are selected in the order in which their modules are instantiated. If this holds, the one can draw the conclusion that logic time in SystemC has an implied third dimension: it is a vector $(t, n, p_{id}) \in \mathbb{N}^3$, and thus simulation events are totally ordered, which makes any simulation deterministic. SystemC's LRM explicitly states: *"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."*

```
#include "systemc.h"

using namespace sc_core;


std::string sharedVariable;

SC_MODULE(chaos1)
{
    sc_in<bool> clock;

    void run()
    {
        while(1)
        {
            wait();
            sharedVariable = "chaos";
        }
    }

    SC_CTOR(chaos1)
    {
        SC_THREAD(run);
        sensitive << clock.pos(); // static sensitivity
    }
};

SC_MODULE(chaos2)
{
    sc_in<bool> clock;

    void run()
    {
        while(2)
        {
```

```
36              wait();
37              sharedVariable = "and destruction";
38          }
39      }
40
41      SC_CTOR(chaos2)
42      {
43          SC_THREAD(run);
44          sensitive << clock.pos(); // static sensitivity
45      }
46
47  };
48
49  SC_MODULE(observer)
50  {
51      sc_in<bool> clock;
52
53      void run()
54      {
55          while(2)
56          {
57              wait();
58              cout << sharedVariable << endl;
59          }
60      }
61
62      SC_CTOR(observer)
63      {
64          SC_THREAD(run);
65          sensitive << clock.pos(); // static sensitivity
66      }
67
68  };
69
70
71
72
73
74  int sc_main(int argc, char *argv[])
75  {
76      sc_clock clock("clock", 1, SC_NS);
77      chaos1 c1("c1");
78      chaos2 c2("c2");
79      observer ob("ob");
80
81      c1.clock(clock);
82      c2.clock(clock);
83      ob.clock(clock);
```

```
84
85      sc_start(2, SC_NS);
86
87      return 0;
88  }
```