Parallel Simulation of SystemC Loosely-Timed Transaction Level Models

Master of Science Thesis

December 4, 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.

Stockholm, December 4, 2016
   Konstantinos Sotiropoulos

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. Section 1.4 describes the research methodology followed. 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 1.5

1.1 Overview

This project follows the work of Björn Runåker[1] [1] 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 [2], 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 magnitude 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.

---

[1]Björn Runåker works as a Platform Application Engineer for Intel in Stockholm: https://www.linkedin.com/in/runaker

- 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   Research Methodology

The presentation of the research methodology, adopted in this work, is influenced by Anne Håkansson's paper titled *"Portal of Research Methods and Methodologies for Research Projects and Degree Projects"* [3]. This work presents a qualitative research on the field of Parallel Discrete Event Simulator development for Electronic Systems Simulation. The novelty of the subject makes qualitative research a necessary step for establishing the relevant theories and experimentation procedures needed by more quantitative approaches. The methodology applied is illustrated in Figure 1. A further explanation of the figure is imminent:
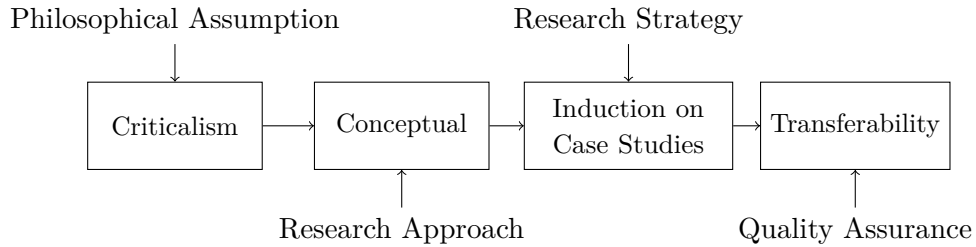
Figure 1: Qualitative Research Methodology

- **Criticalism**: The reality of Parallel Discrete Event Simulator development is being historically determined by the evolution of computational hardware.

- **Conceptual**: Simulator development has not been properly associated with their relevant theoretical understanding: the Discrete Event Model of Computation. Terms like process, time, concurrency, determinism and causality are inconsistently used and usually lack of a proper mathematical definition within a solid framework. The development of the proposed Parallel Discrete Event Simulator is steered by this conceptual exploration. The importance of formalizing concepts with mathematics before development can be seen in the book *"From Mathematics to Generic Programming"* by Alexander Stepanov and Daniel Rose [4],

- **Coded Case studies**: The proposed Parallel Discrete Event Simulator is tested by the implementation of the two case studies.

- **Inductive**: The hypothesis is tested against the successful implementation of the two case studies.

- **Transferability**: The verification of two case studies can only be the basis step of inductive inference. There is still the induction step, that is hoped to be adressed by the proposition of a compiler, that will allow every Loosely-Timed Transaction Level Model to "fit" the proposed Parallel Discrete Event Simulator.

1.5 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 [5].

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** [6].

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 [7] and has been constantly evolving to capture and steer industry practices. Figure 2 presents the form of the Y-Chart found in [5].

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
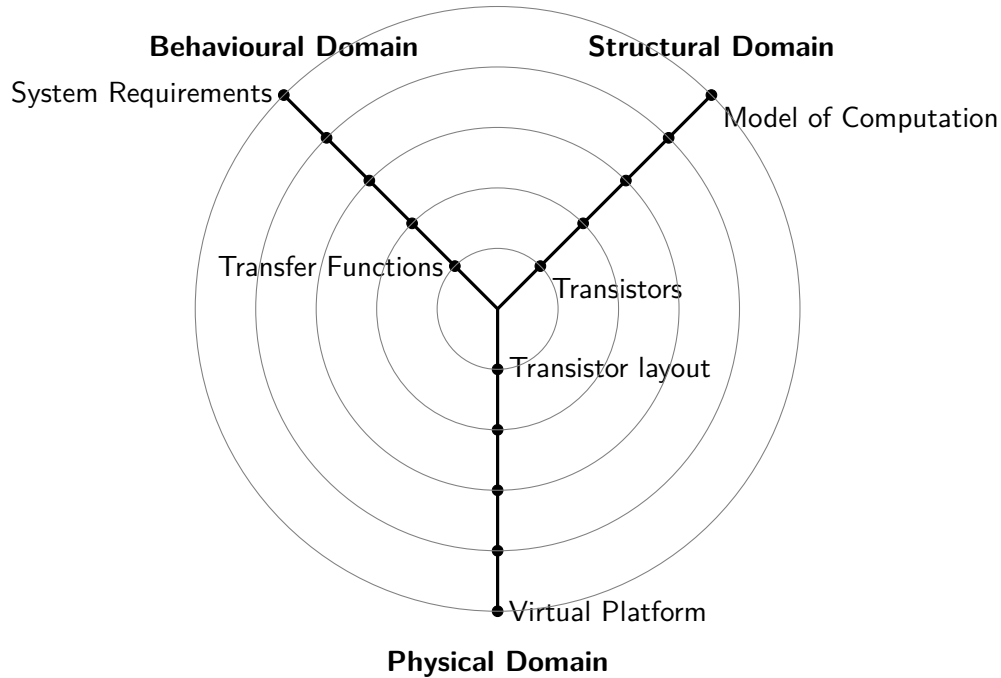
Figure 2: Gajski-Kuhn Y-chart

ways for describing the system: behavioral, structural and physical. These are represented as the three axes, 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 → test/verify → 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 [8]. 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 [9]). 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 [10].

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 the component 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 [11]. 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)** [12].

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**. The set of processes is denoted by $\mathbb{P}$. 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 $i \in \mathbb{N}$. An event denotes a system state change; from the system's perspective, it can be regarded as a mapping $\mathbb{A} \to \mathbb{A}$. $\mathbb{E}$ is a universal set on which processes P_i define a partition:

$\mathbb{E}$ is a partially ordered set under the relationship **"happens before"**, denoted by the symbol $\sqsubset$ [13]. The binary relationship $\sqsubset$, apart from being antisymmetric and transitive, is irreflexive; an event can not "happen before" itself.

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:

$$e_1, e_2 \in \mathbb{E}.e_1 \propto e_2 \implies e_1 \sqsubset e_2 \tag{1}$$

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.
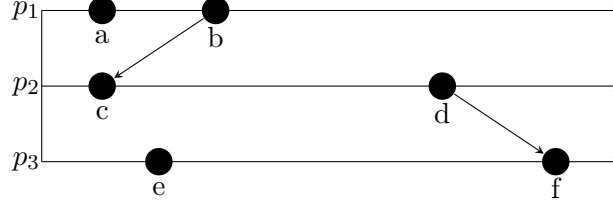

Figure 3: DE space-time decomposition

Figure 3 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 [2].

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 time modeling technique for relating an event with a number, its timestamp, has not yet 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

A realization of the DE abstract machine is called a **Discrete Event Simulator (DES)**. When implementing a DES, 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's environment; for example a x86 Time Stamp Counter (TSC) measuring the number of cycles since reset. **Logic/real time** is defined as a the notion of time in the DES; a **logic time modeling** technique associates an event with a value, which is called its **timestamp**. Since $\mathbb{E}$ is partially ordered and only the sets P_i are totally ordered, one is forced to reach the conclusion that the nature of the DE MoC instigates a **relativistic notion of logic time**. Logic time may be different across processes, at any moment in real time, and it is only through communication that a global perception of logic time can be formulated.

Logic time modeling is deferred to the implementation of the DE abstract machine and is highly depended on the nature of the undelying hardware. Is it **parallel**, where the spatial decomposition

---

[2] 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.

defined in the DE can be preserved? Or is it **sequential**, where space dimensionality must be emulated. The only restrictions DE semantics impose on a logic time modelling technique C are:

$$e_1 \sqsubset e_2 \implies C(e_1) < C(e_2) |Range(C)| = \aleph_0 \tag{2}$$

If a DES can infer a total ordering of $\mathbb{E}$, through a logic time modeling technique, 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 presents the fundamental mechanism behind SystemC's DES: coroutines. Sections 2.3.2 to 2.3.4 give an analytic description of the actions performed in SystemC's simulation environment. An algorithmic description of the simulator's main event loop can be found in Section 2.3.5. The Section is complemented by the code examples found in Appendices A and B.

2.3.1 Coroutines

SystemC's distribution comes with a sequential realization of the DE MoC, referred to as the reference **SystemC simulation engine** [9]. 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[3], 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 of control flow?

2.3.2 The kernel

The **kernel** is the simulation's director [6], 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]:

1. If there are no events in the **global event queue** and the list of runnable processes is empty, it must **terminate** the simulation.

2. It sorts the global event queue according to timestamp order.

3. It possesses a global perspective over logic time: **global time** advances according to the timestamp of the event (from the global event queue) last triggered.

4. When the list of runnable processes has been depleted, it is his duty to trigger the next, according to timestamp order, event. It first checks whether there are events in the **delta**

---

[3]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.

[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.

notification queue. Triggering these events do not advance global time. It then checks the global event queue.

5. 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.

6. 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 implementation-defined. An example of such a situation can be found in Appendix B.

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

2.3.3 Modeling Time

Logic time can be represented as a vector [6] $\in \mathbb{N}^n$ where $n \in \mathbb{N}$. This time modeling technique is referred to as **superdense time** [6]. Every event is associated with a vector; in other words, every event has a timestamp. Ordering of events comes as a lexicographical comparison between timestamps.

SystemC explicitly defines logic time as a vector (t, n). Although, as demonstrated in Appendix B, there is an implied third dimension.

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. The choice of using the term "superdense" for this logic time modeling technique can now be understood: between any two events e_1, e_2, with modeled real time values t_1, t_2, $\exists e_3$, such that $timestamp(e_1) < timestamp(e_3) < timestamp(t_2)$. 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**.

To avoid quantization errors and the non-uniform distribution of floating point values, SystemC internally represented logic time 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 timing 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 Event Notification and Process Yielding

Events in SystemC are realized as instances of the class `sc_event`. Processes perform event notifications, by calling either of these variations of the `sc_event.notify` method:

- `notify(sc_time t)`: (Scheduled occurence) The process adds the event to the global event queue. All sensitive processes will become runnable when the kernel triggers the event.

---

[6] This terminology is not consistent across literature, for example the term **dense** [14] may also imply that logic time $\in \mathbb{R}$ or $\mathbb{Q}$. By Cantor's *"diagonal count"*, $|\mathbb{N} \times ... \times \mathbb{N}| = \aleph_0 < |R|$. The terms **superdense** and **dense** in this case are semantically different.

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

- `notify()`: (Immediate notify) The process signals a flag within the kernel. All sensitive processes in the waiting list are moved to the runnable list, at the next context switch.

- `notify(SC_ZERO_TIME)`: (Delayed occurence) The process adds the event to delta notification queue. All sensitive processes in the waiting list are moved to the runnable list, after the runnable list becomes empty.

Yielding is explicitly stated by a calling a variant of the `sc_module.wait` method. The most important are:

- `wait()`: The process remains in the waiting list, until events in its sensitivity list are triggered.

- `wait(sc_time t)` Before yielding, the process adds a newly created event in the global event queue, with timestamp = `t + global_time`. It also becomes sensitive to this event.

- `wait(sc_event e)` Before yielding, the process modifies its sensitity list, so as to include `e`

2.3.5 SystemC's Main Event Loop

What follows is an algorithmic description of SystemC's main event loop.

Algorithm 1 SystemC's event loop (kernel's perspective)

 1: **while** scheduled events exist **do**   ▷ Global clock progression loop
 2:     order events in global event queue
 3:     trigger the event with the smallest timestamp
 4:     advance global time
 5:     make all sensitive processes runnable
 6:     **while** runnable processes exist **do**   ▷ Delta cycle progression loop
 7:         **while** runnable processes exist **do**   ▷ Immediate notifications loop
 8:             run a process
 9:             trigger all immediate notifications
10:             make all sensitive processes runnable
11:         **end while**
12:         trigger all delta notifications
13:         make all sensitive processes runnable
14:     **end while**
15: **end while**

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. In unit 2.4.2 we give an overview of prior art in the field of PDES in ESLD. is an event safe to execute

2.4.1 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 [15]. A trivial realization of the above scenario is depicted in figure 4. 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.
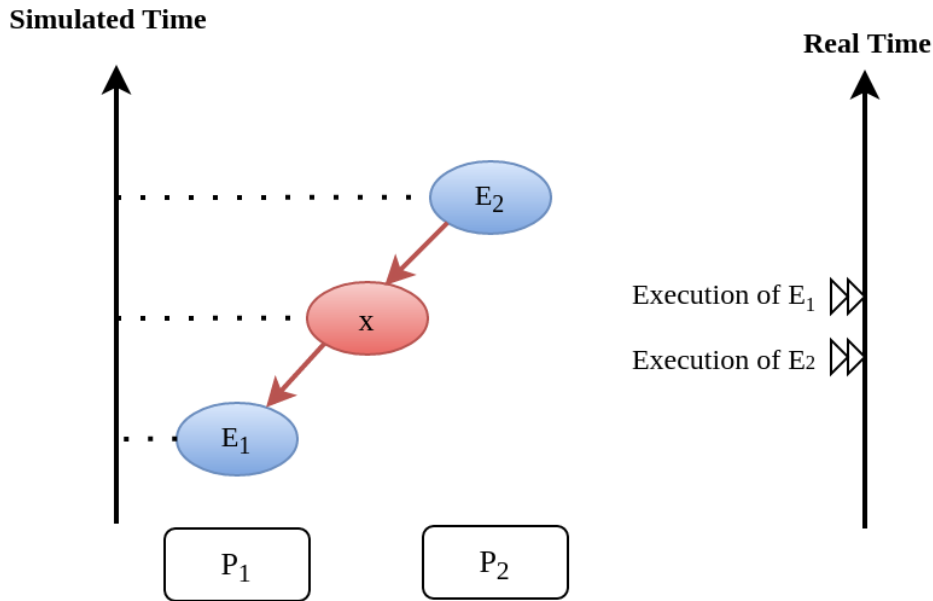
Figure 4: Causality error caused by the sharing of the system's state variable x by P_1 and P_2.

The second and most difficult to deal with source of causality errors is depicted in figure 5. 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 [15]? 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
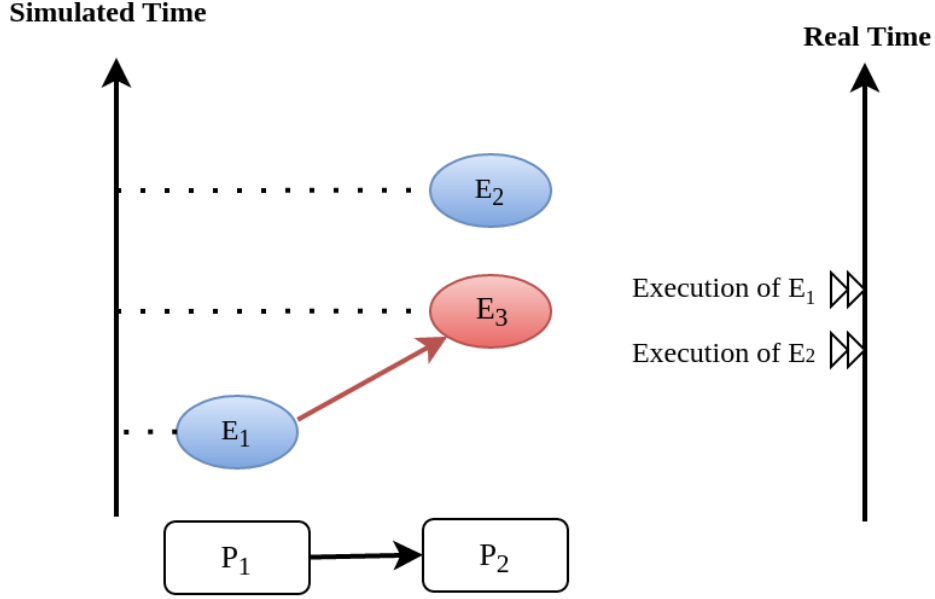
Figure 5: Causality error caused by the unsafe execution of event E_2 (adopted from [15]).

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** [16]. 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 [15].

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 [17] .

2.4.2 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 [18]. 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 later sections, this approach will bare no fruits in the simulation of TLM Loosely Timed simulations, since delta cycles are never triggered [19]. Therefore, we switch our interest in **Out-of-Order PDES (OoO PDES)** [20]; allowing each process to have its own perception of simulated time, determined by the last event it received.

The most important: RISC: Recoding infrastructure for SystemC [21].

Miscellaneous: SystemC-SMP [22] SpecC [23], although the latter is not meant for SystemC. sc\during [24]

2.5 Message Passing Interface

Section 2.5.1 presents the rationale behind choosing MPI, as the means for achieving spacial decomposition, in the proposed OoO PDES. In unit 2.5.2 and 2.5.3 we present the semantics of the Message Passing Interface (MPI) communication primitives. This Chapter is complemented by Appendix C, where the reader can experience MPI's elegance, by means of an example implementation of the pipeline pattern. Please note that the term process in this section refers to the Operating System concept.

2.5.1 Rationale

Message Passing Interface 3.0 (MPI) was the preferred implementation framework for the proposed OoO PDES. Events are modeled as structured messages, and communication as message passing. The rationale behind this choice can be summarized as follows:

- The ease of expressing process communication, that leads to improved readability and maintainability, when compared to other process manipulation APIs (e.g. POSIX)

- Scalability. Tons of it. Any computing device or cluster with Internet Access, from a Raspberry Pi to Tianhe-2, is more than welcome to participate in the simulation. If the MPI runtime environment is configured properly, the software developer may remain agnostic about the exact communication fabric (e.g. shared memory, TCP/IP, DAPL).

- High performance. Prior to version 3.0, MPI was deemed a bad choice for applications confined in shared memory nodes. Threading APIs (e.g. OpenMP), or hybrid approached were a more favorable choice. With the introduction of MPI 3.0, shared memory regions, for conducting communication apart from message passing, can be exposed to processes.

2.5.2 Semantics of point-to-point Communication in MPI

MPI is a message passing library interface specification, standardized and maintained by the Message Passing Interface Forum [25]. It is currently available for C/C++, FORTRAN and Java from multiple vendors (Intel, IBM, OpenMPI) [25]. 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 [26].

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.

2.5.3 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 [26]:

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 eventually occur. 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.

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. In Appendix C the reader can find an example use case for this communication mode.

3 Out of Order PDES with MPI

The goal of this chapter is to present the process synchronization algorithm that will be applied in the proposed OoO PDES. In Section 3.1 and 3.2 we present the conservative synchronization algorithm known as **Chandy Misra Bryantt** (CMB). In Section 3.3 a pseudocode description of the CMB is demonstrated. The pseudocode incorporates MPI communication primitives.

3.1 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)** [27] [28]. Historically, it has been the first of the family of conservative synchronization algorithms [15]. 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 partioned amongst the system's processes. Execution is reactive; it is sparked by an event and produces further events and side-effects (changes in the system's state variables). Each process keeps its own perspective of logic time through a counter. The counter advances according to the timestamp of the last event selected for execution.

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 an unbounded [8] 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 [27] [28].

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 **blocking** a process until each of its incoming links contains an event. All the above are demonstrated in Algorithm 2. The synchronization algorithm is realized as a process' main event loop.

Algorithm 2 Process event loop, without deadlock avoidance

1: **while** process time < some T **do**
2:     **Block** until each incoming link contains at least one event
3:     select event M, with the **smallest** timestamp across all incoming links.
4:     set process' **counter** = timestamp(M)
5:     **execute** event M
6:     **communicate** resulting events over the appropriate links
7: **end while**

3.2 Deadlock Avoidance

The naive realization of the process' event loop presented in Algorithm 2 leads to deadlock situations, like the one depicted in Figure 6. The links placed along the outer loop are empty (dashed lines),

---

[8]The system description is quite similar to that of another MoC called *"Kahn process networks"* [6], which also uses unbounded FIFOs as a channel communication mechanism. The difference is qualitative: the DE MoC incorporates timing semantics. A Kahn process network is **untimed** by definition.

thus simulation has halted, even though there are pending events (across the links 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, and thus increase the implementation complexity of the simulation environment. For the context of this thesis, a distributed mechanism is more favorable. What follows is the presentation of a distributed mechanism for overcoming these situations, referred to as the **null-event deadlock avoidance** [29].
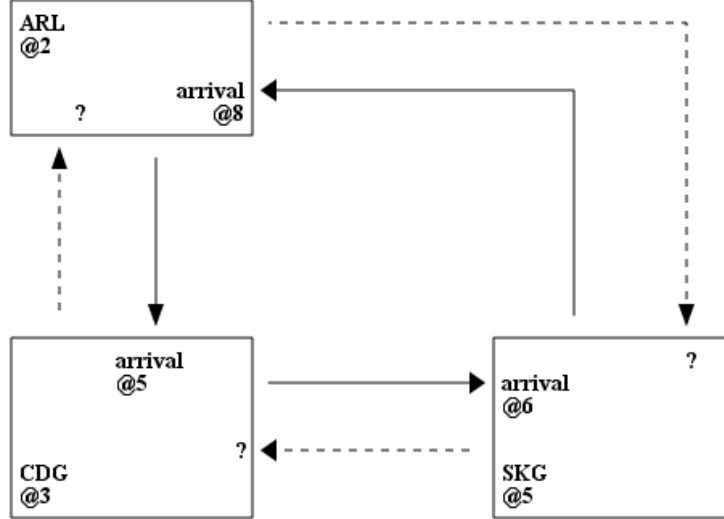
Figure 6: Deadlock scenario justifying the use of Null messages in the CMB

Figure 6 demonstrates an air traffic simulation, where the airports (ARL, CDG and SKG) constitute the simulation processes. The events exchanged between the airports represent flights (the time unit being arbitrary). 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 $(P \times P) \to \mathbb{N}$. By selecting the distance between every airport to be 3 time units, one can deduce the following: If SKG is at time 5, then ARL or CDG should not expect any flight arriving from SKG before time 8.

The simulation is deadlocked: all of the airports contain an empty link and therefore, according to Algorithm 2, they must block. At deadlock, the counter values for each airport are: (ARL,2), (SKG,5), (CDG,3). The intuition behind any technique, that could break the deadlock, should rely on the followig observation: if CDG knew that SKG is at time 5, then it could be able to accept the incoming fligh from ARL, without breaking causality.

To "communicate" this information, SKG could create a special kind of event, a **null event** that does not represent a flight. Its timestamp should be 8 (counter+lookahead) and the event should be placed on all of SKG's outgoing links. With this null event, SKG is informing the other airports about its time perspective. A null event is still an event, so CDG would acknowledge it during the selection phase, and thus would be 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, that would in turn unblock ARL. It is evident that the deadlock situation has been resolved, at the expense of flooding the communication links with null events.

The modified, for deadlock avoidance, algorithm is described in Algorithm 3. The important facts one must keep in mind with this deadlock avoidance mechanism are:

- The logic time counter of a process is still determined by the last event selected for execution.

- Null events are created when a process updates its logic time counter.

- 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.

Algorithm 3 Process event loop, with deadlock avoidance

 1: **while** process clock < some T **do**
 2:   **Block** until each incoming link FIFO contains at least one event
 3:   Remove event M with the smallest timestamp from its FIFO.
 4:   Set process' clock = timestamp(M)
 5:   **React** to event M
 6:   **Communicate** either a null or meaningful event to each outgoing link with timestamp = clock + lookahead
 7: **end while**

3.3 MPI Realization of CMB

Listing 4 is a pseudo code, sketching out the CMB synchronization algorithm with null event deadlock avoidance, using MPI's communication primitives. The mechanism should be incorporated in a process' main event loop. It is quite obvious that the concept of an event has been reduced to a simple data structure, with the timestamp being the most important field. Much like SystemC, logic time modeling is an implied vector (t, n, l): t is the value of a process' counter, n (delta) and l are implied by, the event's position in the links' FIFO and the process' rank, respectively.

Algorithm 4 CMB Process event loop in MPI

---

   while process clock < some T **do**
 2:   post a MPI_Irecv on each incoming peer process
     post a MPI_Wait: block until every receive has been completed
 4:   save each message received in a separate, per incoming link, FIFO.
     identify message M with the smallest timestamp
 6:   set counter = timestamp(M)
     process message M
 8:   post a MPI_Issend to each outgoing link L with timestamp = counter + Lookahead(myRank, recvRank)
     end while

---

4 Methodology

This chapter is a synoptic presentation [9] of the case studies constructed for the evaluation of the proposed OoO PDES. Section 4.1 presents an airtraffic simulation, following the example presented in Section 3.2. The Section is complemented with Appendix D

4.1 Case Study 1: Airtraffic Simulation

The topological arrangement of the airports is determined at compile time. For example, the 3 airport topology described in Figure 6 is demonstrated in Appendix D Every airport is given a schedule of departures and the system's computational objective is to create a log of departures/arrivals per airport.

An airport's main event loop is demonstrated in Figure 7. The series of communication primitives presented in Algorithm **??** have almost the same aggregate effect as the collective communication primitive `MPI_Neighbor_Allgather`. There is one big difference however. By employing this collective operation, communication and execution confront to the their simple definition, presented in Section 2.2, where both input and output require/produce one event.

4.1.1 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.

4.2 Case Study 2: Cache-coherent Multiprocessor

4.2.1 TLM 2.0 translation

---

[9]The source code for the case studies is publicly accessible in the following github repository: `https://github.com/kromancer/Thesis`.

```
                              ┌──────────────────┐
         yes                  ╱  !readyToTerminate ╲
  ┌──────────────────────────▶╲                   ╱
  │                            ╲                  ╱
  │                                    │
  │                                    ▼
  │                         ┌──────────────────────────┐
  │                         │  MPI_Neighbor_Allgather(...) │
  │                         └──────────────────────────┘
  │                                    │
  │                                    ▼
  │        no               ┌──────────────────────────┐
  │ ──────────────────────▶ │  Event e = links.nextEvent() │
  │                         │     counter = e.tstamp()    │
  │                         └──────────────────────────┘
  │                                    │
  │                                    ▼
  │                               ╱          ╲        yes       ┌──────────────┐
  │                              ╱ !e.isOutbound() ╲ ─────────▶ │  updateLog() │
  │                              ╲               ╱              │  sendbuf = e │
  │                               ╲           ╱                 └──────────────┘
  │                                    │                              │
  │                                    ▼                              │
  │                               ╱          ╲        yes       ┌──────────────┐
  │                              ╱ !e.isInbound() ╲ ─────────▶ │  updateLog() │
  │                              ╲               ╱              └──────────────┘
  │                               ╲           ╱                       │
  │                                    │                              │
  │                                    ▼◀────────────────────────────┘
  │                               ╱              ╲
  │                              ╱ links.isAnyEmpty() ╲
  └──────────────────────────────╲  || !e.isOutbound() ╱
                                   ╲                  ╱
```
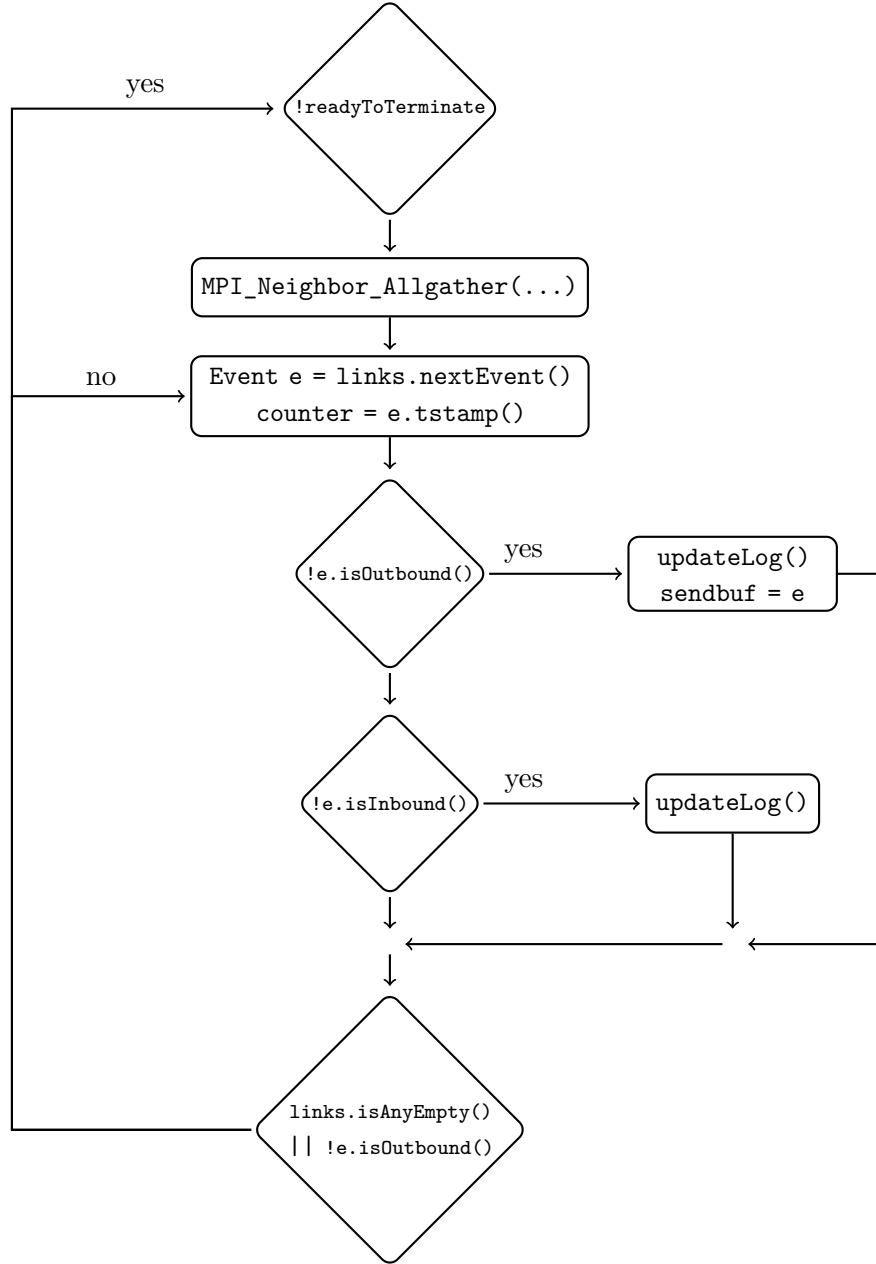
Figure 7: Case Study 1: Airport's event loop (in `Process::run()`)

5 Analysis

5.1 Time Complexity

The following assumption is made: the time complexity of a deterministic DES (Section 2.2.4), is of the form $\mathcal{O}(f(|\mathbb{E}|))$. In the proposed OoO PDES:

$$|(E)| = f(|\mathbb{P}|, \min\{lookahead(P_i, P_j)\}_{i,j \leq |\mathbb{P}|}). \tag{3}$$

In simpler terms:

- The total number of simulation events is highly sensitive to the number of null messages.

- The number of null messages produced is proportional to the minimum lookahead value across the system.

- In the worst case, the minimum lookahead will be 1. This introduces a qualitative shift in time complexity, which now becomes: $\mathcal{O}(f(t_{end} - t_{start}))$ with t_{end} and t_{start} being the timestamps of the first and last simulation events.

This well established empirical observation [29] for simulators dependent on the CMB algorithm, has been confirmed.

5.2 Monotonicity of Communication

In the DE MoC, any a communication function possessing the property $e_1 \sqsubset e_2 \implies g(e_1) \sqsubset g(e_2)$ is called **monotonic**. In the context of the proposed PDES, it can be easily proven that any communication function that uses a lookahead which is only influenced by (P_i, P_j) is monotonic. The communication functions that are used in the case studies are monotonic.

And here comes a reasonable question: What happens if the lookahead function is not only influenced by (P_i, P_j). For example, consider the following situation: at an airport, an airplane's departure is followed by the departure of a faster airplane. Both the flights are destined for the same target airport. The faster aircraft is meant to arrive at the target airport sooner. Using the visual understanding of the DE MoC from Section 2.2.3, the situation is captured in Figure 9.
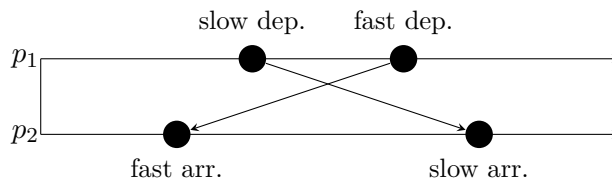

Figure 8: Non-monotonic communication in the DE MoC

How can a DES cope with non-monotonic communication? A naive approach would be to define a static execution schedule: first p_1 and then p_2. The naivety of the approach lies in the following self-contradiction: why run a simulation if you already know its outcome?

The proposed PDES can not handle such a situation. Timestamps in p_2 's incoming link are not placed in an increasing timestamp order by p_1 and thus p_1 is bound to face a causality error: its local time will advance backwards. It is therefore evident that the only way to tackle this situation, while keeping the CMB synchronization mechanism, is to transform the model in a way that all

communication functions are monotonic. To identify the needed transformation, one must take a closer look on the nature of the imbalance that causes non-monotonicity. The concept of airplane speed was introduced, and speed is nothing more than a backdoor for time: alas, time has managed to break loose from the confining cage of logic time modeling; he demands explicit introduction into the system as a process!

The needed transformation is depicted in Figure 9. The weights over the links denote the lookahead (L for lookahead). The two airport processes, p_1 and p_2, no longer communicate directly with each other. They rely on process "time" to advance the flight through space.
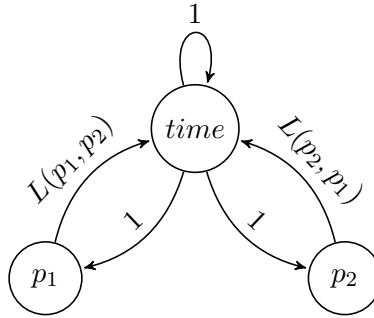

Figure 9: Non-monotonic transformation using the CMB synchronization algorithm

Process "time" self communicates, thus forms a causal loop. Process "time" is a **clock**, with the term "clock" being now formally defined as any process with a feedback loop governed by a computation function without a fixed-point. An intuitive understanding of this definition can be easily conceived, if one thinks of a digital logic NOT gate that has its output wired to its input.

It is quite clear that this system has the worst-case time complexity of the proposed OoO PDES: the minimum lookahead is 1. A DES that always produces worst-case behavior is called a **step simulator** [10].

For SystemC's DES the situation is quite similar. The transformation can be described in many ways, but the core idea remains the same: there is a need for an `sc_clock` instance, which is nothing more than a predifined process, functioning in the same way as the "time" process.

---

[10]Handling non-monotonic communication, while having $\mathcal{O}(f(|\mathbb{E}|))$ time complexity, is an interesting property, a DES might possess. If this situation has not been explicitly identified, the author would like to claim the name `"Sander-Ungureanu-Sotiropoulos (SUS)"`, to describe this property.

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 [11]. 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 [13] suffices in describing the important concepts emanating (e.g. causality, concurrency and determinism).

- **Topology mapping** To the best of our knowledge 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.

6.2 Limitations

- The theoretical description of the DE MoC in section 2.2 is far from complete. Since the DE MoC is considered as an abstract machine, there should be a proof that would indicate its equivalence with some form of a Turing machine. In the same spirit, Section 5.2 assumes that a deterministic DES is equivalent to a deterministic Turing machine, without presenting a 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 [12] and Dr. Jakob Engblom [13] 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.

---

[11] 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/

[12] 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.

[13] 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.

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:

- For the critical task of analyzing the model (identifying the processes and the links between them), ForSyDe SystemC's approach could be mimicked [30]. 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 [31].

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 [14] 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.

---

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

- 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.

7 References

[1] 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

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

[3] 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

[4] A. A. Stepanov and D. E. Rose, *From Mathematics to Generic Programming*, 1st ed. Addison-Wesley Professional, 2014. ISBN 0321942043, 9780321942043

[5] 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

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

[7] 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

[8] 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

[9] 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

[10] ——, *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

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

[12] 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

[13] 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

[14] 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

[15] 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

[16] ——, "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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] citation needed, *citation needed*.

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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   SystemC: Producer Consumer

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:

```cpp
#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.

```cpp
//*************
// 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
```

```
11      int    wi; // write index
12      int    numReadable, numRead, numWritten;
13      // For notifying Producer and Consumer
14      sc_event Ev_dataRead;
15      sc_event Ev_dataWritten;
16  public:
17      // Constructor
18      explicit Fifo(int _size=16):
19          sc_prim_channel(sc_gen_unique_name("thefifo"))
20          {
21              size = _size;
22              buf = new char[_size];
23              reset();
24          }
25      // Destructor
26      ~Fifo(){ delete [] buf; }
27      int  numAvailable() const { return numReadable - numRead; }
28      int  numFree() const { return size - numReadable; }
29      void reset() { free=size; ri=0; wi=0; }
30      void write(char c);
31      char read();
32      void update();
33  };
```

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.

```
1   // Blocking write implementation
2   void Fifo::write(char c)
3   {
4       if (numFree() == 0)
5           wait( Ev_dataRead );
6       numWritten++;
7       buf[wi] = c;
8       wi = (wi+1) % size; // Circular buffer
9       free--;
10      request_update();
11  }
12  // Blocking read implementation
13  char Fifo::read()
14  {
15      if (numAvailable() == 0)
16          wait( Ev_dataWritten );
17      numRead++;
```

```
18        char temp  = buf[ri];
19        ri = (ri+1) % size; // Circular buffer
20        free++;
21        request_update();
22        return temp;
23    }
```

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.

```
1  // Update method called in the UPDATE phase of the simulation
2  void Fifo::update()
3  {
4      if (numRead > 0)
5          Ev_dataRead.notify(SC_ZERO_TIME);
6      if (numWritten > 0)
7          Ev_dataWritten.notify(SC_ZERO_TIME);
8      numReadable = size - free;
9      numRead = 0;
10     numWritten = 0;
11 }
```

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.

```
1  class Producer: public sc_module
2  {
3  public:
4      sc_port<fifo_write_if> out;
5      sc_in<bool> clock;
6      void run()
7          {
8              while(1)
9              {
10                 wait(); // wait for clock edge
11                 out->write(1);
12                 cout << "Produced at: " << sc_time_stamp() << endl;
13             }
14         }
15     // Constructor
16     SC_CTOR(Producer)
17         {
18             SC_THREAD(run);
19             sensitive << clock.pos();
```

```
20              }
21      };
22
23
24      class Consumer: public sc_module
25      {
26      public:
27          sc_port<fifo_read_if> in;
28          sc_in<bool> clock;
29          void run()
30              {
31                  while(1)
32                  {
33                      wait(); // wait for clock edge
34                      char temp = in->read();
35                      cout << "Consumed at: " << sc_time_stamp() << endl;
36                  }
37              }
38          SC_CTOR(Consumer)
39              {
40                  SC_THREAD(run);
41                  sensitive << clock.pos();
42              }
43
44      };
```

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

```
1       int sc_main(int argc, char *argv[])
2       {
3           sc_clock clkFast("ClkFast", 1, SC_NS);
4           sc_clock clkSlow("ClkSlow", 500, SC_NS);
5
6           Fifo fifo1;
7
8           Producer p1("p1");
9           p1.out(fifo1);
10          p1.clock(clkFast);
11
12          Consumer c1("c1");
13          c1.in(fifo1);
14          c1.clock(clkSlow);
15
16          sc_start(5000, SC_NS);
17
18          return 0;
19      }
```

B SystemC: Non-Determinic yet Repeatable

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."* One could device the following terms to describe this situation: **non-deterministic yet repeatable** or **pseudo non-deterministic**.

```cpp
1  #include "systemc.h"
2
3  using namespace sc_core;
4
5
6  std::string sharedVariable;
7
8  SC_MODULE(chaos1)
9  {
10     sc_in<bool> clock;
11
12     void run()
13     {
14         while(1)
15         {
16             wait();
17             sharedVariable = "chaos";
18         }
19     }
20
21     SC_CTOR(chaos1)
22     {
23         SC_THREAD(run);
24         sensitive << clock.pos(); // static sensitivity
25     }
26  };
27
28  SC_MODULE(chaos2)
29  {
30     sc_in<bool> clock;
31
32     void run()
33     {
34         while(2)
```

```
35              {
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  }
```

C MPI: The Pipeline Pattern

Much like an assembly line, the pipeline pattern is useful when the problem at hand can be understood/modeled as a series of chained actions. The magic happens by overlapping execution and communication. Just think of how cars are manufactured. The following example simulates the daily routine of an assembly line worker in a car factory.

As with any MPI program, the reader must keep in mind that, unless explicitly specified, all of the processes will get a copy of the **same executable**. So as to differentiate control flow, the process must acquire its unique ID from the MPI runtime environment.

```cpp
#include "mpi.h"
#include <iostream>
#include <climits>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int myId, workerLeft, workerRight, numWorkers;
    bool isFirst(false), isLast(false);
    int completedTask(INT_MIN), nextTask(INT_MIN), taskAtHand(INT_MIN);

    // Register with the factory's administration
    MPI_Init(&argc, &argv);
    // What is my id
    MPI_Comm_rank(MPI_COMM_WORLD, &myId);
    // How many workers in the factory
    MPI_Comm_size(MPI_COMM_WORLD, &numWorkers);
```

Each assembly line worker must know who is on the left and who is on the right.

```cpp
// Who is on my left
if( myId != 0)
    workerLeft = myId - 1;
else
{
    // The first worker has access to the raw materials
    isFirst = true;
    taskAtHand = 0;
    nextTask = 0;
}

// Who is on my right
if( myId != numWorkers-1)
    workerRight = myId + 1;
else
{
    isLast = true;
```

```
18        workerRight = myId;
19    }
```

For each worker there is a task at hand, a new task coming from its left and the task he completed. The communication is carried out by the conveyor belt, so it is not her/his concern. She/He just places the completed task on the belt, takes the next task and continues working. While the task at hand is being computed, communication is happening on the background.

After some initial delay, proportional to the size of the pipeline (the number of workers), cars are produced at a constant rate!

```
1     // Daily routine (or lifelong slavery?)
2     while(1)
3     {
4         MPI_Request signals[2];
5         MPI_Status  statuses[2];
6         // Communicate but don't block
7         if (!isLast)
8             MPI_Isend(&completedTask, 1, MPI_INT, workerRight, 1, MPI_COMM_WORLD,
   ↪ &signals[0]);
9         if (!isFirst)
10            MPI_Irecv(&nextTask, 1, MPI_INT, workerLeft, 1, MPI_COMM_WORLD,
   ↪ &signals[1]);
11
12        // Simulate Work
13        taskAtHand++;
14        sleep(1);
15
16        // Work is done, prepare for communication
17        completedTask = taskAtHand;
18        taskAtHand    = nextTask;
19
20        // This is meaningful only to the last worker
21        if( completedTask == numWorkers)
22            std::cout << "Car is ready!" << std::endl;
23
24        if (isFirst)
25            MPI_Wait(&signals[0], &statuses[0]);
26        else if (isLast)
27            MPI_Wait(&signals[1], &statuses[1]);
28        else
29            MPI_Waitall(2, signals, statuses);
30    }
31
32    MPI_Finalize();
33    return 0;
34 }
```

An open source and gratis implementation of the MPI API is the Open MPI. Under the

assumption that an MPI implementation has been installed, the steps for compilation and execution look like:

```
1  mpic++ pipeline.cpp
2  mpirun -n 4 a.out
```

D   Case Study 1: Airport Topology

The following header file describes the airport topology presented in Figure 6

```cpp
// FILE: topology.hpp
#ifndef TOPOLOGY_HPP
#define TOPOLOGY_HPP
#include <string>
enum  Airport {SKG, ARL, CDG};
const int lookahead[3][3] = { {0,3,2}, {3,0,1}, {2,1,0} };
const Airport airports[3] = {SKG, ARL, CDG};
const std::string airport_to_string[3] = { "SKG", "ARL", "CDG"};
const int indegree[3] = {2,2,2};
const int sources[3][2] = { {1,2}, {0,2},  {0,1} };
const int sourceweights[3][2] = { {1,1}, {1,1}, {1,1}};
#endif
```

The information is deserialized at runtime by `MPI_Dist_graph_create_adjacent`, which is responsible for creating the airport's "neighborhood". Therefore, the fundamental communication primitive used, `MPI_Neighbor_allgather`, will have all the necessary information to perform the broadcast and collect (allgather) operation.

```cpp
// FILE: process.cpp
#include "process.hpp"
#include "topology.hpp"

Process::Process(int rank)
    : rank(rank),
      counter(0),
      num_neighbors(indegree[rank]),
      no_inbound_flights_in_links(true)
{
    MPI_Info info;
    MPI_Dist_graph_create_adjacent( MPI_COMM_WORLD, \
                                    num_neighbors, \
                                    sources[rank], \
                                    sourceweights[rank], \
                                    num_neighbors, \
                                    sources[rank], \
                                    sourceweights[rank], \
                                    info, 1, &my_neighbors);
...
}
```