A CO-SIMULATION ENVIRONMENT FOR MIXED SIGNAL, MULTI-DOMAIN SYSTEM LEVEL DESIGN EXPLORATION

by

David Kent Reed

BS, University of Pittsburgh, 2002

Submitted to the Graduate Faculty of the

School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

University of Pittsburgh

2004

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

David K. Reed

It was defended on

July 20, 2004

and approved by

James T. Cain, Professor, Electrical Engineering

Raymond R. Hoare, Assistant Professor, Electrical Engineering

Thesis Advisor: Steven P. Levitan, Professor, Electrical Engineering

A CO-SIMULATION ENVIRONMENT FOR MIXED SIGNAL, MULTI-DOMAIN SYSTEM LEVEL DESIGN EXPLORATION

David Kent Reed, MS

University of Pittsburgh, 2004

This thesis presents a system-level co-simulation environment for mixed domain design exploration. By employing shared memory IPC (Inter-Process Communication) and utilizing PDES (Parallel Discrete Event Simulation) techniques, we examine two methods of synchronization, lock-step and dynamic. We then compare the performance of these two methods on a series of test systems as well as real designs using the Chatoyant MOEMS (Micro-Electro Mechanical Systems) simulator and the mixed HDL (Hardware Description Language) simulator from Model Technology, ModelSim. The results collected are used to ascertain which method provides the best overall performance with the least overhead.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

First, I sincerely thank Dr. Steve Levitan, my advisor, for the awesome opportunity to work with him. His guidance and discipline has transformed me, in many ways, into a harder worker and more versatile engineer. Secondly, I would like to thank Dr. Ray Hoare for his time and effort in evaluating my work and for being an important part in the completion of my thesis. I extend a special thanks to Dr Tom Cain for his help and guidance throughout my graduate experience, including being a part of the completion of my thesis work.

I extend my thanks and everlasting friendship to the many friends I have made in the Keystone CAD group, both past and present. Jason, Mike, Craig, Jose, Majd, and Amit: you all have been incredibly supportive and good friends. I especially thank Jason for his diligent help in helping me during this project. I also would like to thank Gustavo for being a good friend and partner in the struggle for attaining our Master's degrees. I wish all of you the best of luck in all of your endeavors.

I would not be here if it had not been for the love and support of my parents, sister, and the rest of my family. They have been there in thick and thin. I specially thank my mother for being the best help and critic in the world and my father for being the supporter of my many conquests throughout life. I also thank my sister for being a great sounding board and sharing those midnight trips to eat and shop. I also thank my grandmother for her prayers that helped me through everything, as well as the rest of my grandparents, wherever they may be. I also thank my new family, Mariya and Josef, for their support and help throughout the past 3 ½ years.

Lastly, though he is not family, I would like to thank Scott for being the brother I never had and his support since we first met at Engineering 11.

Lastly, I cannot begin to find the words and emotions to thank my wife, Juliya. She has been my biggest fan and the source of my sanity throughout everything. Without her I would truly be lost and still looking for my way. I can only hope that this work and everything that I will do can prove to her that I am worthy of everything she has done for me. I can only say to you, Julz, Я тебя люблю.

1.0 INTRODUCTION

There have been significant advances in the area of mixed signal technologies in terms of digital and analog circuits as well as optical and mechanical devices. Using System-on-a-Chip (SoC) concepts and fabrication mechanisms such as Multi-Chip-Modules (MCM), these four technologies can now be integrated together permitting a new realm of system development. Such developments as Micro-Electro-Mechanical Systems (MEMS) and Micro-Optical-Electro-Mechanical Systems (MOEMS) are possible and thus advancing computing technology (1, 2).

With the above in mind, there have been efforts put forth in designing the verification tools used for these systems, however, mostly pertaining to a specific domain. Due to the complexity of these systems, it is difficult to verify that all components are operating correctly. This difficulty lies in the fact that there are not many tools that provide end-to-end simulation of MEMS or MOEMS (2).

Nevertheless, the need for a system wide simulation tool that can prove digital, analog, mechanical, and optical components that all function together is evident. Also, such a system simulation must be as fast and accurate as possible. One way to provide these features is to create a co-simulation environment that is capable of handling the digital, analog, mechanical, and optical technological domains. Each domain is handled by the most fitting simulator available and then coupled via an efficient co-simulation architecture. This in turn provides a system simulation environment needed to perform analysis and verification.

A complete system level simulation encompasses and examines many aspects required to verify that every component will function and operate with other pieces of the system. Unlike circuit or digital simulation and likewise mechanical or optical modeling and analysis, all of which focus on a particular domain, system level simulation must process every element, regardless of its physical domain, simultaneously.

In examining modern co-simulation technology, we have found that much of the current research on system simulation is in the realm of embedded system design (3, 4, 5). This involves the co-simulation of both hardware and software systems in a framework or toolset. Hardware/software co-verification is limited to RTL or behavioral hardware description language (HDL) simulation and an interface to an instruction set simulator (ISS) (5). As a result, design exploration and verification is reduced to digital and software based analysis and does not typically account for analog circuitry, mechanical modeling, or optical analysis.

Optical and mechanical simulation typically involves performing both static as well as dynamic analysis. Static simulation is used to measure mechanical tolerance, crosstalk, as well as power and insertion loss. Dynamic simulation analyzes noise, data streams, and bit error rates (1). The simulations just mentioned, whether static or dynamic, are typically performed in a simulator created for a certain physical domain (2). Examples of such tools are ANSYS Finite Element Analysis (8) for mechanical modeling and RSoft for optical analysis (9).

There is an attempt to unify the simulation environment for the physical domains found within MOEM systems with a common toolset and language. The core concept is to provide a universal language that has the capability to describe the behavioral functionality of every domain found in a system. Once this description is created, a software simulator is utilized to perform a system level simulation (4). This is seen in the many SystemC simulators and language flavors available (3). This concept, coupled with the use of HDL's and other modeling techniques, provides a method for performing system level simulation (3, 4).

When considering system level description languages such as SystemC, there are other languages available for modeling system level components. These languages are based on the hardware description language (HDL) varieties available for digital system design. Mixed Signal HDL's provide a designer with the same language semantics already found in the digital system realm, however, with the added benefit providing facilities to describe other components (2). Examples of the Mixed Signal HDL's are VHDL-AMS and Verilog A. VHDL-AMS is an extension to the VHDL 93 language. VHDL-AMS provides the facility to describe mathematical expressions as well as maintaining a larger built-in type definition for real voltages, lumens for optics, and force measurements for mechanical simulations (10). Similarly, Verilog-A is built on the similar constructs of Verilog, providing the same modeling capabilities as VHDL-AMS (10).

Utilizing a higher level object-oriented approach, C++ is also a viable option for system level modeling. For purposes of this thesis as well as an example of the using C++, Chatoyant provides an environment for simulating components, independent of what their behavior is, in a true mixed-signal, multi-domain platform. Chatoyant, built on and utilizing Ptolemy (1, 2), expresses the components within a particular system as C++ objects, each encapsulating data types and behavior unique to a particular device. Chatoyant takes advantage of the different simulation domains within Ptolemy (i.e. Dynamic Data Flow, or DDF, and Discrete Event, or DE) to perform analysis of optical, mechanical, and electrical components at both the macro and micro level (1, 2). This tool will be discussed in more detail in Chapter 2.

One potential problem that arises from the existence of the simulation and system analysis tools previously mentioned, with the exception of Chatoyant, is the inability to handle a particular physical domain, albeit optical, mechanical, analog, digital, etc. Another potential problem is that there is no direct connection to another simulator that performs a different analysis that would bridge this gap in a complete system simulation. In such a case, a co-simulation, or real-time link between two or more simulators, becomes a solution. A design is typically partitioned into the pieces that will simulate in the appropriate simulator. Then an interface is defined between these simulators created based on the signals that will connect from one domain to another.

A simple example for such a need for co-simulation is in a digital and analog system design. If the digital portion of the design was created using VHDL or Verilog, and the analog sub-system was specified in the form of a SPICE netlist, a standard HDL simulator could only execute the VHDL description and only a SPICE simulator could simulate the netlist. Therefore, to perform a complete simulation, concurrently, a co-simulation interface must be used between the HDL and SPICE simulator. Some factors to consider in this case is first, how the two simulators will remain synchronized throughout the simulation time and second, how information is translated between the two physical domains (3). These considerations are discussed in detail in Chapter 3. The concept of co-simulation and the potential need for it brings us to the purpose of this research. The following section will detail techniques, results, and limitation of the work accomplished. It constitutes the statement of the purpose for this thesis

1.1 STATEMENT OF PURPOSE

The purpose of this research is to examine and create an efficient co-simulation interface between Chatoyant and ModelSim. As mentioned in the previous section, Chatoyant is a simulation environment for performing MOEM system level design exploration and simulation. ModelSim is a commercially available mixed-language HDL simulator. This research addresses the lack of HDL simulation support found in Chatoyant for digital systems by bridging Chatoyant to ModelSim for a real-time simulation interface.

1.2 STATEMENT OF WORK PERFORMED

To implement the co-simulation interface needed to address Chatoyant's lack of HDL simulation support, many approaches were researched and examined. Among the different techniques considered, the realm of Parallel Discrete Event Simulation (12) was studied in depth. Since both simulators were discrete event by nature, the issue of time synchronization was readily solved by many techniques found in the domain of PDES. This thesis provides the details of the theory used to implement the interface It also discusses the trade-offs between two common synchronization algorithms used in PDES. The remainder of the thesis focuses on the background, theory, implementation, and testing of the co-simulation environment. The following section describes what each chapter discusses in more detail.

1.3 LIMITATIONS OF WORKED PERFORMED

The co-simulation interface presented in the previous section is based entirely on discrete event (DE) simulation. This process, though one of the more detailed simulation techniques for time-based simulation, also has one of the highest overheads in terms of scheduling processes, when compared to cycle-based or instruction level simulation (7). Also, since both of the simulators used in the co-simulation simulate using discrete event simulation, the interface between them must transfer individual events. As seen in Chapter 5 with the test systems, this works well for sparsely distributed events. However, for systems that have a clock or periodic signal that cross the interface, there is a large performance penalty due to a high volume of inter-simulator event traffic. The future work section of Chapter 6 describes potential solutions to this problem and possible methods of modifying this particular co-simulation interface to obtain reduced inter-simulator event traffic and lower synchronization overhead.

1.4 CHAPTER DESCRIPTIONS

Chapter 2 covers the background on the technology used in the co-simulation interface described by this thesis. This includes a look into ModelSim, the commercial mixed language digital simulator, Ptolemy, a multi-domain simulation framework from the University of California at Berkeley, and Chatoyant, a complete simulation environment for performing MOEMS system simulation, built on top of Ptolemy. This provides the foundation of the toolsets used in the co-simulation system described in Chapter 4.

Chapter 3 provides a look into the different components needed for the co-simulation environment. Topics covered by this chapter include what simulation techniques are applicable to the co-simulation interface's design, described in this thesis, as well as a more detailed look into parallel discrete event simulation concepts. The chapter concludes with a discussion on how we measure the performance of the co-simulation interface.

Chapter 4 describes the details behind the architecture of the co-simulation environment between ModelSim and Chatoyant. This includes a detailed look into the algorithms and setup of both simulators for the two types of synchronization that we describe in Chapter 3. Finally, there is a discussion about the automated system generator that creates all of the necessary files needed to implement the co-simulation environment for a particular system.

Chapter 5 provides the descriptions and results of a series of test systems. These systems provide insight into how the co-simulation environment operates from simple to complex systems and real-world designs. Runtime results such as the number of messages, cycles per second, and system resource utilization are examined for all of these systems, utilizing two different synchronization algorithms. This chapter concludes with a cross comparison of all of the systems and a general conclusion as to which synchronization approach is the overall best.

Chapter 6 summarizes and concludes this thesis, reiterating the general concepts and goals of the research performed. It reviews the systems used to test the interface and provide final remarks about the results collected. Finally we look into what improvements can be made to the co-simulation interface and what is required to make the co-simulation interface a more complete and efficient tool.

A set of appendices contain a list of key definitions used throughout this thesis along with the complete set of runtime measurements that we use in Chapter 5.

2.0 BACKGROUND

The co-simulation interface between Chatoyant and ModelSim utilizes three tools. ModelSim is a commercial product, from Model Technology Incorporated, a subsidiary of Mentor Graphics. We discuss, in more detail, ModelSim in Section 2.1. Chatoyant, a tool for performing mixed signal multi domain system, or MSMD [(1)], was developed at the University of Pittsburgh. Chatoyant was developed to support the modeling and simulation of optical, mechanical, and electrical components found MOEM systems. Chatoyant utilizes the Ptolemy heterogeneous simulation framework for the actual simulation process [(1, 2)].

Ptolemy utilizes an object-oriented programming, or OOP, paradigm to model user designs and components [(7)]. After we discuss ModelSim, we first examine Ptolemy to provide a proper background for Chatoyant. Second, we examine Chatoyant and its contribution in the realm of system-level simulation. We then conclude with a brief look into the need for a co-simulation interface between Chatoyant (ultimately Ptolemy) and ModelSim.

2.1 MODELSIM

ModelSim® works on a single kernel concept which means that all design components, independent of their source language, are compiled into one primary executable object. This executable object provides dynamic links to all compiled design objects. After all components have been compiled by the ModelSim compiler, the top level master design file is loaded, which in turn engages an elaboration procedure. This process links in all external components and loads them into the ModelSim memory space. Coincidentally, at this time all external C applications are linked as well from any foreign language interface (FLI) or programming language interface (PLI) callouts. Likewise, any delay annotation files such as SDF (Standard Delay Format) for back-annotated simulation, are linked and applied. At this point ModelSim begins preparation for simulation.

ModelSim is considered a discrete event simulator (30), which means that all activity within components and between them is treated as individual events with timestamps. In the ModelSim environment, the user can force events on any visible net and then tell the simulator to advance to a stop point. Since ModelSim is a mixed language simulator, it supports of both Verilog and VHDL. The implication of this feature requires ModelSim to simulate Verilog designs differently than VHDL. In Figure 1 we illustrate ModelSim's method of performing simulation for any VHDL design.

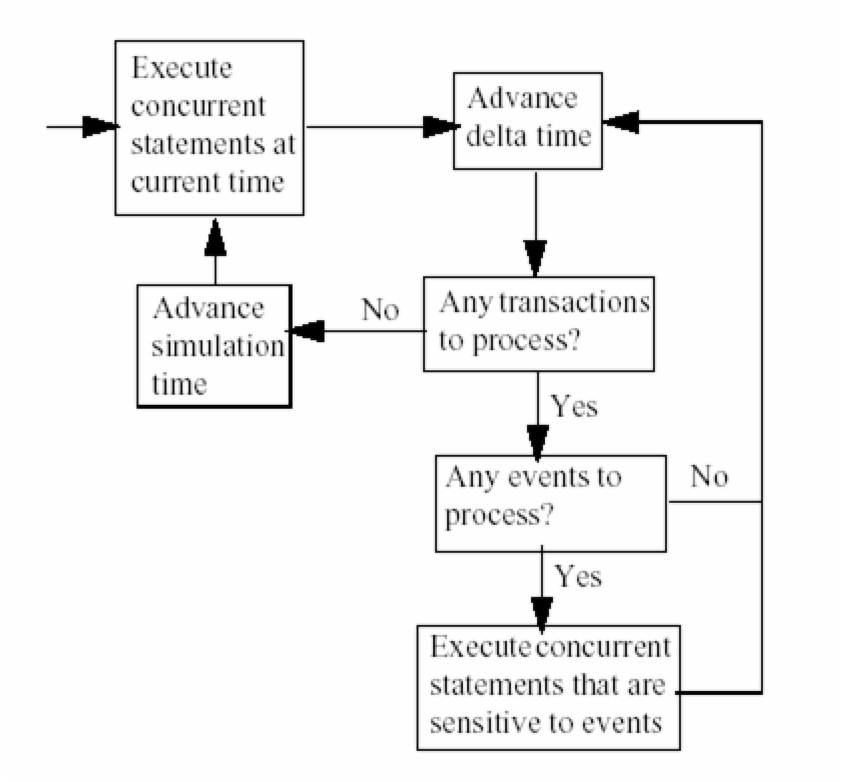

Execute concurrent statements at current time
Advance delta time
Advance simulation time
No
Any transactions to process?
Yes
Any events to process?
No
Yes
Execute concurrent statements that are sensitive to events

Figure 1 Reproduction of ModelSim's VHDL simulation flow using a discrete event methodology [(30)]

This diagram, reproduced from the ModelSim User's Guide, [(30)], shows the flow of every VHDL simulation. First, every concurrent statement at the current time is executed. Next, a delta cycle counter is incremented and any new event transactions are detected. If there are any transactions to be processed, then the simulator checks to see if there any events to be executed. If there are events that need to be executed, they are now processed and the delta cycle is incremented again. If there are no transactions, then the simulator determines that it is safe to advance the simulation clock. This process repeats until the user determined stop time or there is an infinite zero-delay loop detected. The infinite loop problem is the next topic for investigation.

An issue with this flow, as well as many discrete event simulators, is that of zero-delay feedbacks. These are typically defined by user logic that contains a combinatorial feedback without any type of storage. This design issue creates a problem in discrete event simulators since events are constantly scheduled for the same time on the same set of nets, with the possibility of looping in this cycle forever (9, 30). ModelSim deals with this issue by using delta-cycles (30). This feature, as seen on the top of Figure 1, is used to determine if stability is attainable, by cycling through a series of delta iterations until a stable value is determined in a feedback loop. If the loop iterates a preset maximum number of times, the simulator stops and reports a maximum loop iteration error (30).

Verilog simulation, in ModelSim, extends the VHDL simulation flow described above. However, the Verilog Language Reference Manual, or LRM, from the Verilog 1364 standard, places constraints on how simultaneous events are processed (30). Verilog is defined by the 1364 LRM such that the simulator cannot control the order in which simultaneous events are handled (38). Since some designs require that events be processed in a particular order, the expected simulation results will vary from the actual results seen by the user (30). Verilog requires that ModelSim and every other Verilog capable simulator maintain a total of seven event queues. These queues are processed in a particular order, also defined by the Verilog standard. Listed below are the event queues, in order of how they are processed during a simulation, and brief description of what each queue holds, as defined by the Verilog LRM.

- *Active events* – These are the current events being processed and are in an arbitrary order.
- *Inactive event* – These events are events that are waiting for all active events to be processed.
- *Non blocking update events* – These events are similar to concurrent statements in VHDL, and are processed after all inactive events have be processed.

- *Monitor events* – These events are events that are left over from the active, inactive, and non-blocking events executions.
- *Future events* – These are events scheduled to occur in the simulation future time, and are divided into:
    - Future inactive events
    - Future non-blocking events

Zero delay oscillations, by the Verilog standard, are handled in the active event portion of the simulation (30, 38). According to the Verilog 1364 standard, all events are simulated only from the active event queue, and all other queues are used to sort event types and priorities (38). For more on this topic refer to the ModelSim User's Manual (30) and the Verilog 1364 LRM Standard (38).

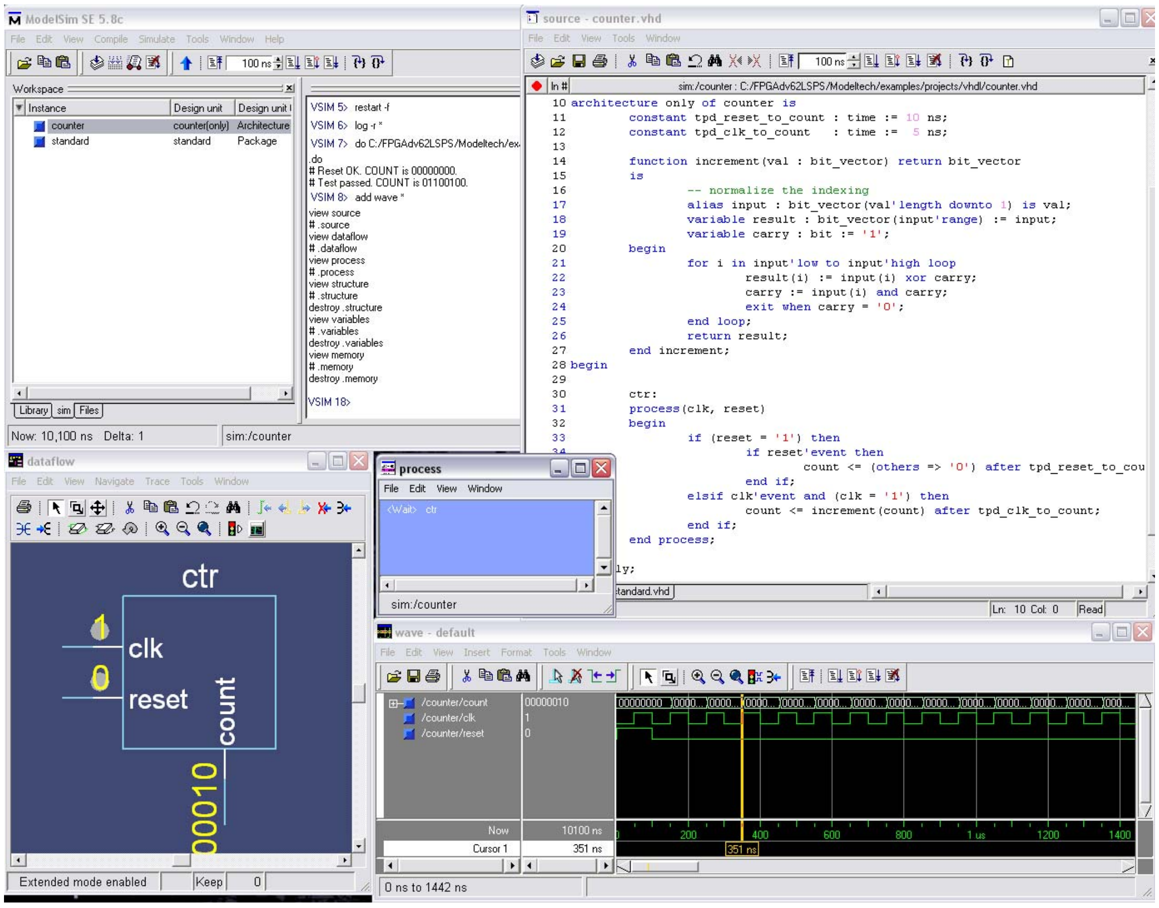
One powerful feature ModelSim provides is the ability to support executing C based object files embedded within VHDL and Verilog designs. This is possible through the Foreign Language Interface, or FLI, used with VHDL or the Programming Language Interface, or PLI, used with Verilog. We will focus on the fundamentals of the FLI since it is used in the implementation of the Chatoyant/ModelSim co-simulation interface.

The FLI, as defined in the ModelSim FLI Reference Manual (31), provides a set of functions that will allow a user to have access to information within the ModelSim environment. These functions will allow the user to traverse a design's hierarchy, obtain and set values of HDL objects such as arrays and signals, obtain information about the ModelSim runtime environment, and within a set of defined limits, control the simulation (31). The FLI code, written in C, can contain operating system dependent functions such as calls to pipes and sockets or shared memory (31). Each FLI program is compiled into a shared object file (.so in UNIX/Linux) or dynamically linked library (.dll in Windows). These programs are then referenced in part of the user's VHDL code, specifically within the architecture of a design unit, with the object file's name and name of the initialization method to call when the simulator begins loading the object file.

The FLI defines two types of methods above the standard functions and procedures found in C. The first of these new types of method is a callback, which defines a function that is registered in the simulator and activated when some criteria, such as simulator restart, are met in the ModelSim environment. For example, there is a callback function that the user can define if the simulation is closing. If the user closes ModelSim, this callback will be activated and executed before the simulator is terminated (31). The second type of function defined by the FLI is a *process* method. This function type is the same as a VHDL process block (27), except it is implemented as a C function (31). The features defined in the FLI reference manual (31) are used in the implementation of the Chatoyant/ModelSim interface. The use of the FLI are discussed in more detail in Chapter 4.

The last topic to discuss about ModelSim is the user interface. ModelSim can operate in both console mode and graphical mode. Both are based on a TCL (Tool Command Language) user interface and the ModelSim GUI makes use of the Tk extension package (30). In terms of user features, ModelSim provides many graphical tools for performing code verification and analysis. The user can select and log all signal and variable activity and view it in a waveform or tabular form. Code can also be debugged in a step-wise fashion similar to a standard software debugger. This is most useful for functional and behavioral simulation. Code performance can be measured using a mechanism based on the GNU code profiler, *gprof* (30), which reports to the user the quantity of time spent in particular segments of the user's code. This is a useful tool for measuring performance in the Chatoyant/ModelSim co-simulation interface. Other tools, such as a coverage measurement feature for performing different types of coverage analysis as well as a waveform comparison feature for measuring mismatches between two simulation runs, exist in ModelSim.

To provide a view as to the look and feel of ModelSim, Figure 2 shows an example snapshot from a ModelSim simulation, courtesy of Mentor Graphics. The design comes from one of the tutorials ModelSim provides as a teaching tool. In this particular example, we show a simple counter running in the ModelSim simulator. We ran this simulation and provided some of the GUI windows found within ModelSim. The upper left window in the screen shows the main ModelSim command console. The upper right window shows the HDL code editor and debug window. The window in the lower right corner displays the dataflow view for the design component selected. The center window shows the current state of all processes found in the current design unit that is selected. The lower right corner shows the waveform view for the simulation.



Instance	Design unit	Design unit t
counter	counter(only)	Architecture
standard	standard	Package

```
VSIM 5>  restart -f
VSIM 6>  log -r *
VSIM 7>  do C:/FPGAdv62LSPS/Modeltech/ex
.do
# Reset OK. COUNT is 00000000.
# Test passed. COUNT is 01100100.
VSIM 8>  add wave *
view source
# .source
view dataflow
# .dataflow
view process
# .process
view structure
# .structure
destroy .structure
view variables
# .variables
destroy .variables
view memory
# .memory
destroy .memory

VSIM 18>
```

sim:/counter : C:/FPGAdv62LSPS/Modeltech/examples/projects/vhdl/counter.vhd

```vhdl
architecture only of counter is
        constant tpd_reset_to_count : time := 10 ns;
        constant tpd_clk_to_count   : time :=  5 ns;

        function increment(val : bit_vector) return bit_vector
        is
                -- normalize the indexing
                alias input : bit_vector(val'length downto 1) is val;
                variable result : bit_vector(input'range) := input;
                variable carry : bit := '1';
        begin
                for i in input'low to input'high loop
                        result(i) := input(i) xor carry;
                        carry := input(i) and carry;
                        exit when carry = '0';
                end loop;
                return result;
        end increment;
begin

        ctr:
        process(clk, reset)
        begin
                if (reset = '1') then
                        if reset'event then
                                count <= (others => '0') after tpd_reset_to_cou
                        end if;
                elsif clk'event and (clk = '1') then
                        count <= increment(count) after tpd_clk_to_count;
                end if;
        end process;

ly;
```

Figure 2 A snapshot of the ModelSim graphical environment from a test design simulation [(44)]

We now conclude our discussion about ModelSim and turn the discussion to the other tools used in the Chatoyant/ModelSim co-simulation interface. This will include a discussion about Ptolemy and Chatoyant.

2.2 PTOLEMY

Our discussion continues about the tools used in the Chatoyant/ModelSim co-simulation interface with an examination of Ptolemy (7). Ptolemy, from the University of California at Berkeley, is the simulation framework that Chatoyant uses, and therefore we must examine how Ptolemy works and what information it provides. Ptolemy uses the object oriented programming (OOP) paradigm and C++ to define classes that describe different parts of a system. A system in this case is what the user of Ptolemy wishes to model and simulate. Within Ptolemy, there are a series of predefined classes, the most general of which is the *domain* (7).

The domain, for the scope of this research, defines the computation model used to simulate a system. There are numerous domains available in Ptolemy. For example, there is the discrete event domain (used in the Chatoyant/ModelSim co-simulation interface), which is used for digital hardware and network simulations. Another domain is the dynamic data flow domain (found in some Chatoyant models), and a subset domain known as the synchronous data flow domain, both of which are targeted for signal processing applications (7, 35).

Every system, regardless of the domain it simulates in, is comprised of a set of basic units called blocks (7). These blocks, contain user defined code that performs some action, in a method called *go()* (7). The block contains other methods such as *startup()*, which defines what the block must do when a simulation begins, and *wrapup()*, which defines what the block must do when a simulation completes (7).

Typically, a block's *go()* method defines an action based on input stimulus from a Porthole (7). Systems in Ptolemy are created by a series of blocks which connected to another by means of the portholes. Portholes are defined as standard interface by which blocks communicate to one another (7). A domain will define the mechanism for transporting information from porthole to porthole through a scheduler. The data sent through the portholes is defined as a stream of particles (7). Particles also adhere to the OOP style found in Ptolemy and can be primitive data types such as floating point numbers or integers or more complex objects, as seen in the case of Chatoyant. Two other important classes that assist in the connection and transfer of particles between portholes is the *Geodesic* class which establishes the connection between two portholes and the *Plasma* class which acts a garbage collector when a particle has been consumed by the destination porthole (7). In Figure 3, we show a diagram of the flow between two blocks, with portholes, and particles being transported between them.

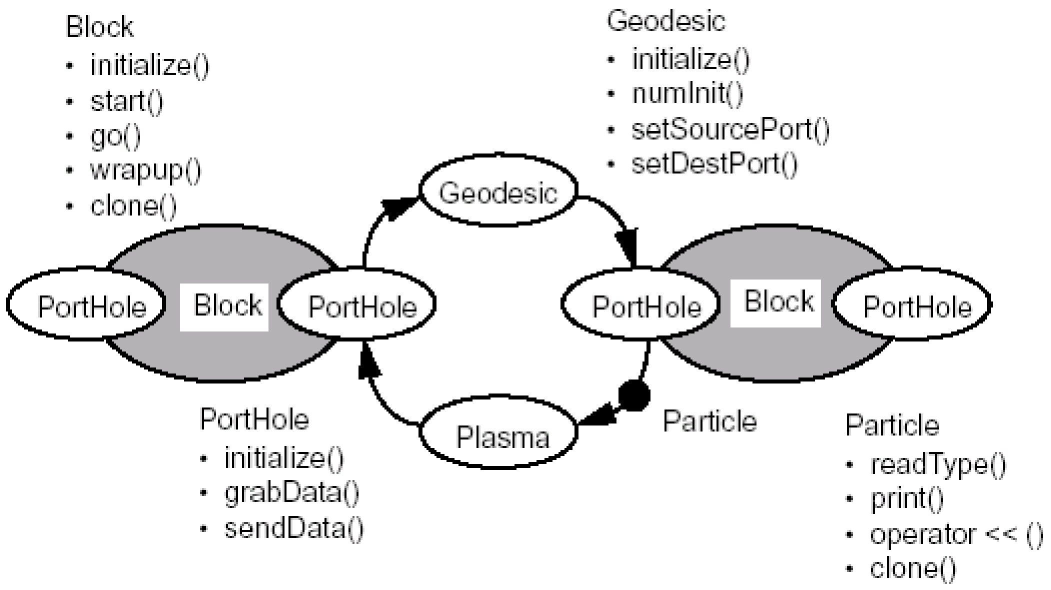


Figure 3 How blocks connect through portholes and how particles are transferred between them in Ptolemy (7)

As seen in Figure 3, each block contains a series of methods that the user must define. These methods perform a specific operation, defined in more detail in (9). Any data that requires transmission to other blocks is done so through the porthole object. In the case of sending, the block will call the sendData() method in the porthole instance. Once again, this data can be in the form of primitive floating point numbers or integers or a complex user-defined data type. The porthole will then pass a particle, which acts as a generic wrapper for the data being transmitted, through an instance of the Geodesic object. The Geodesic instance will contain the destination pointer for the particle. The scheduler of the domain these blocks reside in will call the grabData() method of the destination block. The destination block will then process the data and the particle will be collected by the Plasma instance between the two blocks. This will happen for every connection between every block in a system. (7, 9)

Ptolemy also provides support for a hierarchy when building systems. Every block now becomes defined as a star, which is the atomic unit of component description. The next level of abstraction available in Ptolemy is a collection of Stars called a Galaxy. A Galaxy, models system behavior (through a collection of Stars) and can have portholes the same as Stars. A Galaxy and all of the stars within it, are all confined to a specific domain of simulation (e.g. discrete event or dynamic data flow). All Galaxies or groups of Stars are ultimately contained in a Universe which has similar properties as a Galaxy. A Universe also has the ability of invoking a scheduler and executing a simulation (7).

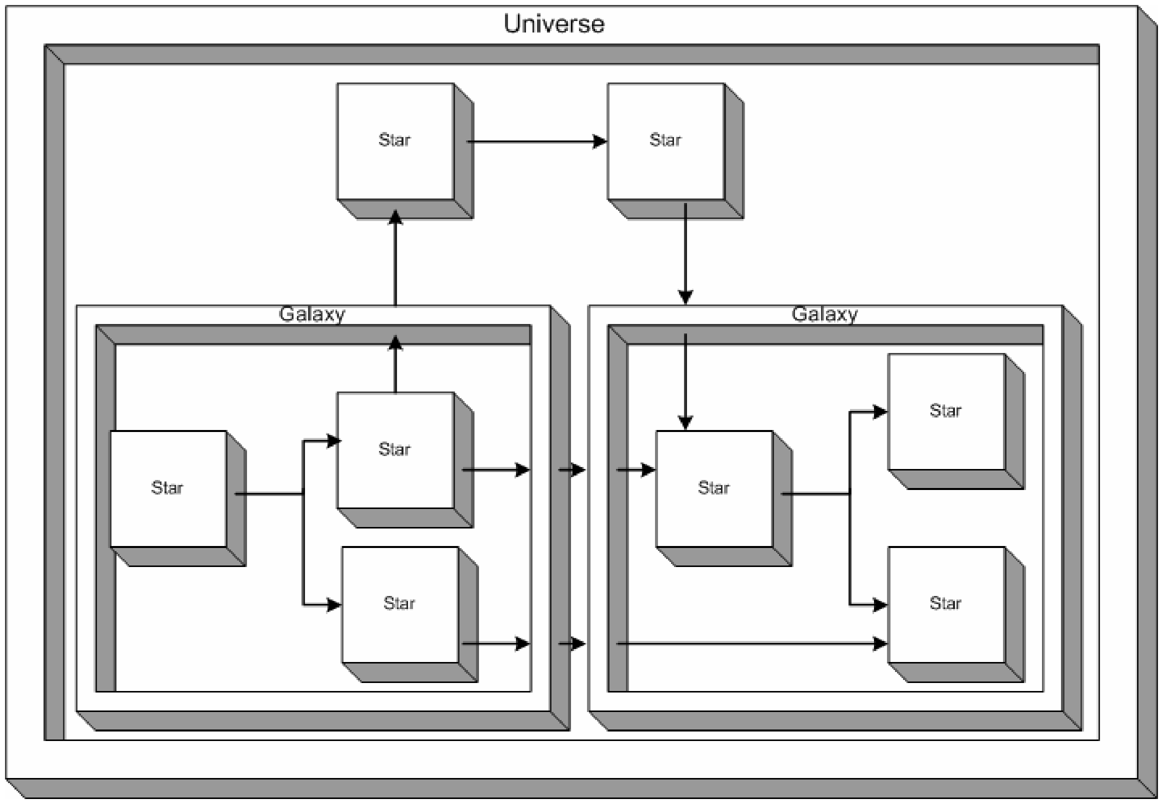


Figure 4 Illustration of a Ptolemy Universe with Stars and Galaxies, derived from (7)

The diagram in Figure 4 shows the how a system can be described using the hierarchical constructs in Ptolemy. First, the stars represent the atomic level block. Each line with an arrowhead represents the connection between the ports of the stars. Next, the borders with the "Galaxy" label represent the description of a particular galaxy. Signals traverse the galaxy border using the same system as seen in the stars using portholes. Finally, the universe border defines the entire system as a whole, acting as a container for all stars and galaxies. The universe will activate the scheduler referenced by all of the stars and galaxies for the domain these items reside in. (7)

As we discuss the methodology of how a system is modeled in Ptolemy, one key item found in a domain and used by a universe, galaxy, and star is the scheduler. The scheduler is responsible for managing the flow of data throughout a simulation as well as the execution of the simulation. There are different schedulers for different domains in Ptolemy (7). For example, there is a scheduler that maintains the order and flow of data particles in the dynamic data flow (DDF) domain. This scheduling methodology is used in signal processing simulations (9). The discrete event (DE) domain requires the scheduler to manage time instead of the flow of data. The DE scheduler maintains an event queue and the global time of the universe during a simulation (9). The DE scheduler must resolve zero-delay oscillations (the same issue handled by ModelSim). Instead of detecting this at runtime, Ptolemy detects a zero-delay loop at compile time, as a universe is loaded, before simulation. More details concerning the different schedulers for the different domains is available in (9).

Ptolemy also provides a feature for allowing simulation between multiple domains. This heterogeneous simulation is limited to a particular combination of domains. First, the main universe of a system is defined, defining the scheduler needed for its operation. For example, a

DE based universe will instantiate the DE scheduler. Secondly, the user can then define a sub-galaxy within the universe that calls upon a different a scheduler, such as the synchronous data flow (SDF) domain and scheduler.

The communication system between both the domains and schedulers is resolved by using a Wormhole object in Ptolemy. A wormhole makes use the EventHorizon interface to resolve the difference in scheduling requirements between the two schedulers as well as any encapsulation differences needed to transfer particles between the two domains. For a more detailed description of the Wormhole, EventHorizon, and domains that work together, see references (7, 9). The importance of mentioning this capability in Ptolemy is relevant for discussing the different components found in Chatoyant. Chatoyant contains objects that utilize domains such as SDF due to their signal processing behavior while other components utilize the DE domain due to the necessity to simulate with a time reference.

The Wormhole, though a powerful mechanism in Ptolemy, is not directly applicable to the Chatoyant/ModelSim co-simulation interface. In order to make use of the Wormhole in the co-simulation interface, either an existing simulation domain, such as DE, must be modified or a new domain needs to be created. If an existing domain is modified, there must be a mechanism implemented to handle both the particle translation to and from ModelSim, as well a substantial modification to the scheduler for it to perform the synchronization. If a new domain is created, it must implement the necessary interfaces, such as the EventHorizon, as well as the scheduler object, in addition to the remainder of the interface to ModelSim. Though this solution could possibly provide a faster co-simulation runtime performance, certain issues such as integration into Ptolemy and ModelSim as well as making a new domain flexible to handle any system, outweighs the potential speedup.

With a broad based knowledge of Ptolemy, Chatoyant builds upon the framework provided by Ptolemy to provide a true Mixed Signal Multi-Domain simulation tool. This includes generic stars for describing optical behavior, piece-wise linear circuit solvers, and other customizable components needed for a complete system simulation.

2.3 CHATOYANT: A MIXED SIGNAL, MULTI-DOMAIN MOEMS SIMULATION TOOL

Chatoyant is a mixed signal and multi domain system simulation tool (1). Chatoyant provides the ability to simulate macro and micro optoelectronic systems. Chatoyant offers a means to perform architectural design and analysis. Components, built as Stars in Ptolemy, represent different models. These models are connected together through signals and use a Chatoyant specific message class. The specific message class contains subclasses for carrying electrical, optical, and mechanical signal information, depending on the type of models that are being connected together (1, 2). The models and the signals passing data between then define the overall the "dynamics" of the system's behavior (37). We turn our focus to the different models available in Chatoyant which encompasses the three physical domains of optics, electronics, and mechanics.

Chatoyant manages component modeling using three different techniques, all described in (1). The first technique, known as the "derived model" approach, is based on modeling a device by its underlying physics. The second modeling technique uses interpolation of empirical measurements take from a fabricated device. This method creates a mapping between a set of inputs and their resulting outputs. The third modeling technique involves utilizing low level simulations to create a reduced order description. The technique utilizes results of a set of simulations or the output of a finite element solver to generate a computational model. These results are used to define a polynomial that is a result of curve fitting or an interpolation of data points for a particular region of operation. Though there are others ways to perform this technique, this was the one chosen in Chatoyant.

From these three modeling techniques, Chatoyant provides four libraries of components. The Optoelectronic library provides models such as a Vertical Cavity Surface Emitting Laser, or VCSEL, Multiple Quantum Well (MQW) modulators, p-i-n detectors. The Optical Library provides models such as lenses, lenslets, and mirrors. The Electrical Library includes CMOS drivers and amplifiers as well as a circuit solver which is capable of providing a simulation that has a proven speed up over SPICE (2) utilizing piece-wise linear simulation techniques described in (1, 2, 37, and 39). The last major library is the Mechanical Library which provides models such as scratch drive actuators and an RF MEMS variable shunt capacitor switch (8) as well as electro-static devices (1).

Many of the components found the four primary libraries of Chatoyant are non-linear, meaning that their behavior is described in terms of non-linear differential equations (39). The simulation of non-linear devices, without any modification or simplification, is difficult and computationally slow, as stated in (2). Chatoyant addresses this problem with the piece-wise linear modeling approach (1, 2, 39).

Piece-wise linear modeling is divided into several steps. First, a device is decomposed into linear and non-linear sub-blocks. The linear sub-blocks typically consist of passive elements such as interconnects or parasitic elements, as described in (2). All non-linear sub-blocks are linearized using the piece-wise techniques described in (2 and 39) and produce a template to be fed into the second stage of the modeling process. This second stage performs Modified Nodal Analysis, or MNA, which produces a mathematical representation of the device being modeled. The linear sub-blocks are directly passed into the MNA process, while the non-linear sub-blocks undergo the linearization procedure mentioned earlier. Both of these sub-blocks are then combined during MNA to produce a linear mathematical expression in the frequency domain.

After MNA process, complex electrical devices have been proven to simulate, in Chatoyant, efficiently and faster than in their original non-linear description, as discussed in (2). The final mathematical expression resulting from the MNA composition provides a linear solver that, when coupled with the constraint that signals must be described in a piece-wise nature, can use a simple transformation between the frequency and time domain (2). Figure 5 illustrates this process from start to finish. We see in this diagram that a device is decomposed into linear and non-linear segments, both of which are fed into the MNA composition process and then produces the linear frequency domain solver. This solver accepts piece-wise linear input in the time domain and produces piece-wise linear output also in the time domain.

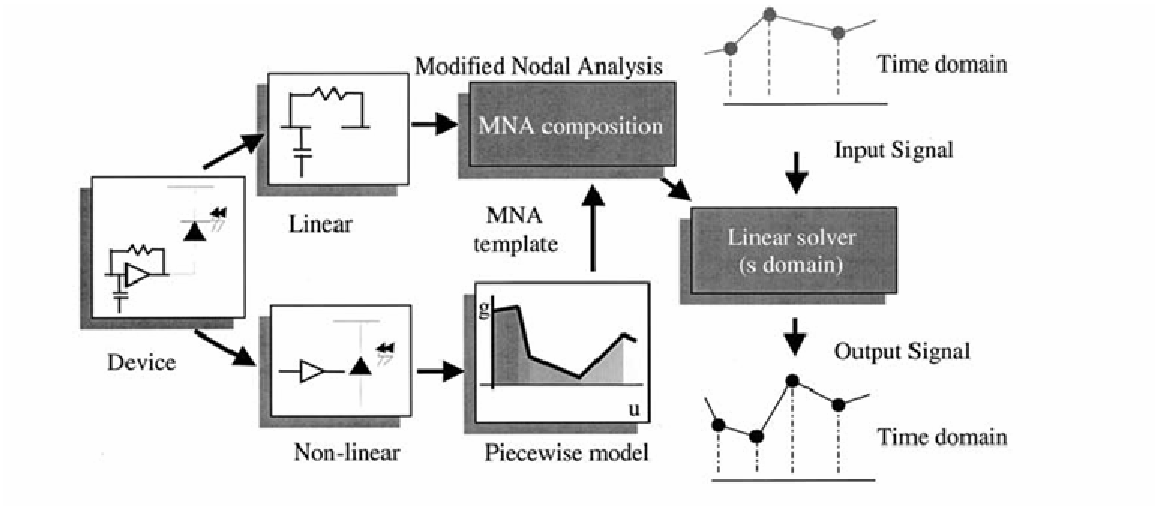


Figure 5 The modeling process, in Chatoyant, for non-linear devices (39)

As described in more detail in (1, 2, 8, 36, 37, 39), the modeling process is applicable to optoelectronics, electronics, and mechanics. Resulting complex devices are modeled quickly providing simulation models that increase the overall simulation speed while maintaining an acceptable tolerance with respect to accuracy of the results, as proven in (1, 2, 8, 36, 37, 39). We now conclude our discussion on Chatoyant and examine the need for the co-simulation interface described in this thesis.

2.4 PROPOSING THE CHATOYANT/MODELSIM CO-SIMULATION

One disadvantage of Ptolemy, and thus Chatoyant, is the lack of support for the simulation of VHDL or Verilog code. This prevents Chatoyant from being a complete end-to-end simulation solution for performing system design exploration. Many systems contain complex digital as well as analog components that couple with optical and mechanical devices. This is often seen in optical computing research (36, 39, 40). Despite the lack of support for direct HDL simulation, the Chatoyant MSMD toolset and the Ptolemy simulation framework provide a promising set of features that can allow the integration of an external HDL simulator. ModelSim provides the mechanisms for communicating to external applications via the FLI. The methods provided by the FLI coupled with a few modifications to Chatoyant and Ptolemy makes a co-simulation interface possible. The remainder of this thesis will explain how the mixed-language HDL ModelSim simulator interfaces to the Chatoyant/Ptolemy toolset.

The following chapter provides a more in depth analysis of the mechanics needed to build the co-simulation interface. We discuss the basic theory behind discrete event simulation and more specifically, parallel discrete event simulation. It then becomes of interest to look into how this theory can be applied to co-simulation, as well as how signals are translated from one physical domain to another, and finally how to measure the performance of the co-simulation interface based on the theory behind discrete event simulation.

3.0 SIMULATION CONCEPTS AND CONSIDERATIONS

In this chapter, we discuss the theory and application of discrete event simulation as it applies to the co-simulation interface between Chatoyant and ModelSim. First, we must define a simulation domain which (presented in (7)) is a computational model for how components in a given simulation interact with one another. One possible simulation domain could be data flow, which simulates according to the flow of data between components. Though a data flow model of simulation has no concept of simulation "time" (7), it does track the events based on the order of data being produced and consumed by components. Data flow models of computation are efficient, though they are more suited to signal processing simulation (7). The discrete event (DE) simulation domain governs the interaction between objects by means of events that change the objects' internal state at some given time (7, 11). The discrete event domain provides a robust model for simulating most systems, however, at the cost of slower runtimes. This cost is seen especially in systems that fit better within a data flow simulation domain (7). Since systems containing electronic components, especially those of a digital nature, frequently measure activity over time, we assume the use of the DE simulation domain for the remainder of the thesis.

In the context of co-simulation, where more than one simulator is executing cooperatively, parallel discrete event simulation (PDES) becomes an important subject to discuss. This realm of discrete event simulation deals with distribution and synchronization of a simulation over multiple event processing entities. Time synchronization is of key importance when considering PDES (11). Therefore, it is the first concept to be examined in detail.

The second concept to be examined is how events cross the co-simulation interface. As we will see in the context of the Chatoyant/ModelSim co-simulation interface, this will imply a translation from one physical domain to another (i.e., analog to digital). The main issue involved with the translation process is how events of a particular type are converted to the correct type and value. The different performance metrics and how they measure the operation of the co-simulation interface is the last topic of this chapter. These concepts are important and are later applied to the operation of the co-simulation interface.

3.1 DISCRETE EVENT SIMULATION FUNDAMENTALS

We divide the discussion of Discrete Event Simulation, or DES, into two sub-topics, as seen in (12). First, we examine sequential DES as well as its limitations. Secondly, we look at parallel DES, or PDES, and specifically discuss conservative and optimistic approaches to asynchronous PDES and the implications of both.

Prior to our initial examination of PDES, it is necessary to define a few key terms. First, a common element among all types of PDES is the scheduler (11, 12). Regardless of the number of schedulers in a system, the main objective of the scheduler is to manage time and event order (12). Although this is a physical process typically executing on a processor in workstation, the scheduler is most commonly visualized as a time wheel (12), as seen in Figure 6.

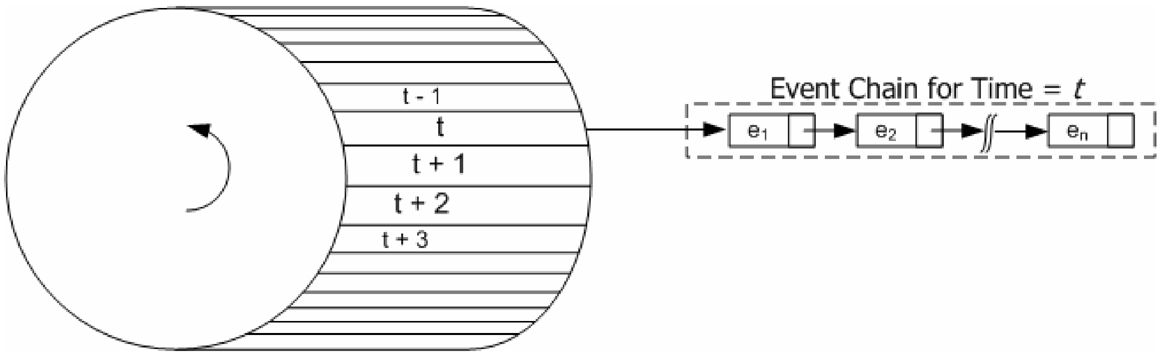


Figure 6 Representation of a Discrete Event Scheduler as a timing wheel, adapted from (12)

As seen in the figure above, the wheel represents a collection of distinct units of time, called time steps (12). As the wheel rotates in a counter clock-wise fashion, time progresses to the next point. At each time stamp, there is a set of events that have been scheduled, taking the form of a linked list in Figure 6. These events are sent to one or more Logical Processes, or LP, defined as a computational block (12), that performs some operation. This operation has the ability to produce new events for the current or a later time step. This wheel is implemented as a priority queue, as seen in (12). As new events arrive from the LP's, the scheduler will sort in the priority queue the new events by the earliest time stamps first. As soon as all activity has ceased for a given time step, the scheduler will proceed to the next time step (9).

After the scheduler issues events, some other process or processes consume these events. A single process is sometimes referred as *logical process*, or LP, as defined in (11, 15, 17, 18). An LP will process a series of incoming events, for a given timestamp, and compute a set of new events for some period of time later (15, 18), by means of a state-change function (19). An example of an LP is a block or star in Ptolemy that consumes events and produces output events (7, 9). We will see, in Chapter 4, that there are two simulators with their own definition of an LP's, one being

Chatoyant with the stars being the LP's, and the other being ModelSim with an LP being the different components loaded into the simulation kernel from a user design. Also, both of these simulators maintain their own simulation time and event queue, independent of one another.

Figure 7 provides a graphical view of the general concept of an LP. In this diagram we illustrate the functionality of consuming events and calculating new events based on some state function. As an event comes into the LP, it is used to lookup which state value is going to be used. In gate-level simulation, this is typically in the form of a gate ID number (12). Using Ptolemy, events occur in the form of a message being consumed at the porthole of a star and then referenced within the *go()* method with in the star (7,9). Once the state is retrieved, the incoming event value is used with the current state to determine if there is state change. This task is typically performed by a lookup table for gate-level simulation (12) or through a user defined process (19), as seen the case of the piece-wise linear solver in Chatoyant. If there is a change in the value of an output from an LP, a new event is created, with a timestamp of explicitly when the event occurs based on delay calculation unique to the user's application (see (12) for examples). This event is sent back to the scheduler and the LP will continue with its execution. It should be noted that this diagram is our own generalization of a logical process.

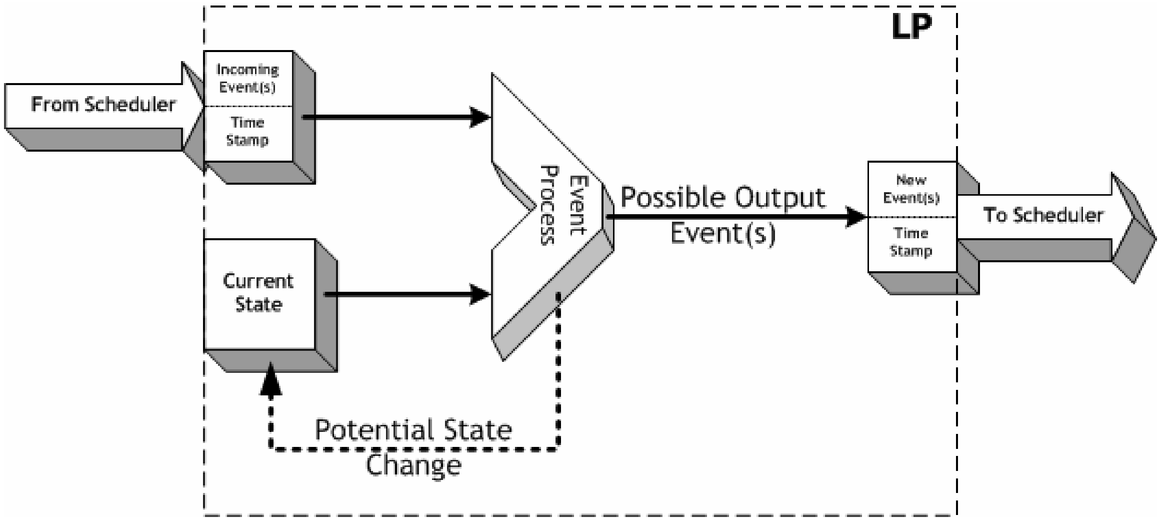


Figure 7 Generalized LP block diagram

With an understanding of the discrete event simulation process, we can now discuss how this works in the context of PDES. In the case of PDES, there are multiple LP nodes (12) and one or more schedulers, depending on the method of synchronization used between those LP nodes. Figure 8a (top box in the diagram) shows the case of sequential DES, such as seen in a DE based universe in Ptolemy. There is a master scheduler and *n* number of LP nodes, with all event traffic routing through the scheduler. Figure 8b (bottom box in the diagram) illustrates an example of asynchronous PDES. In the case seen in Figure 8b, there are now multiple simulation nodes, each having their own local scheduler with on or more local LP's (12). Event traffic is between the local LP's or between the other simulation nodes (12).

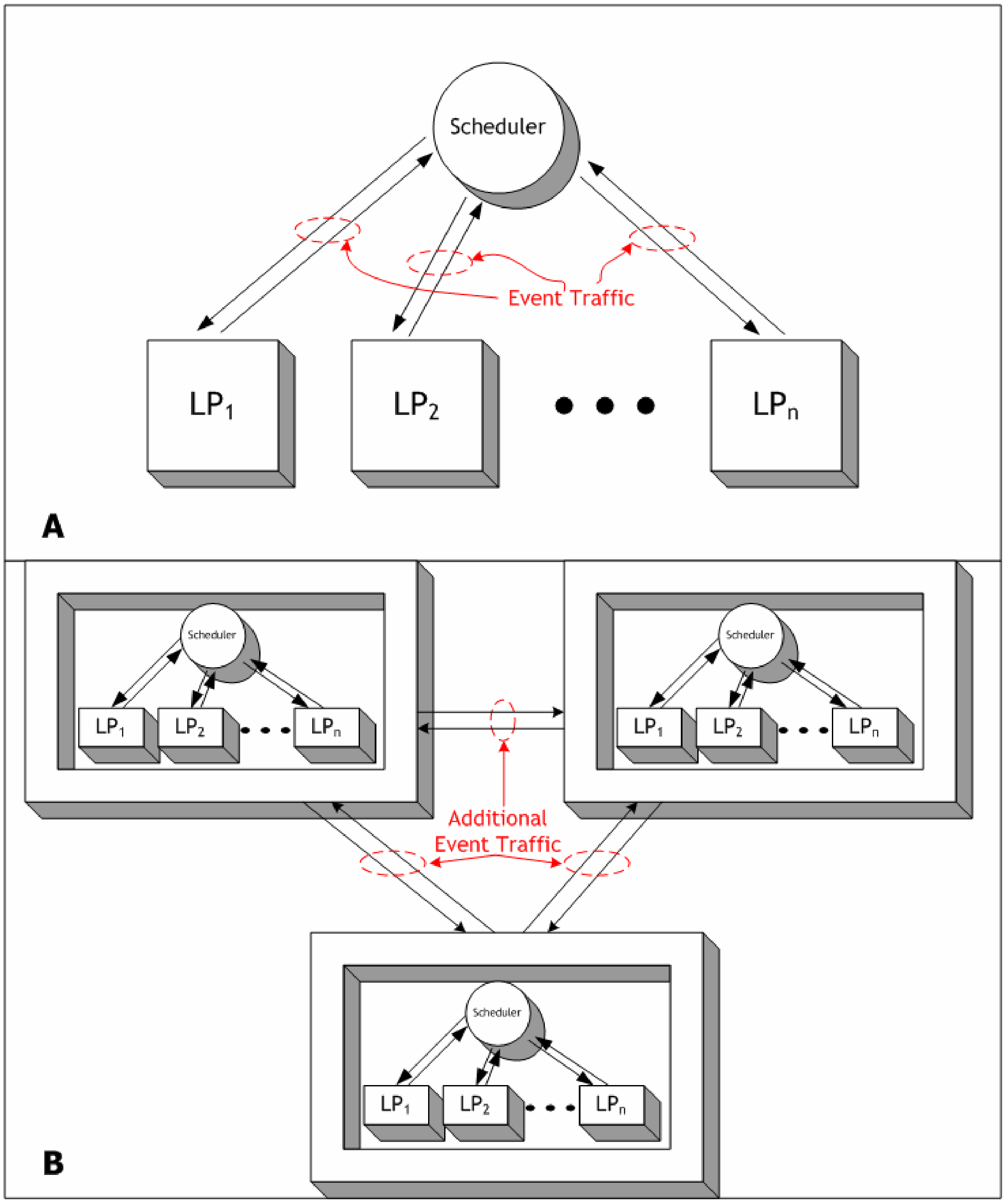
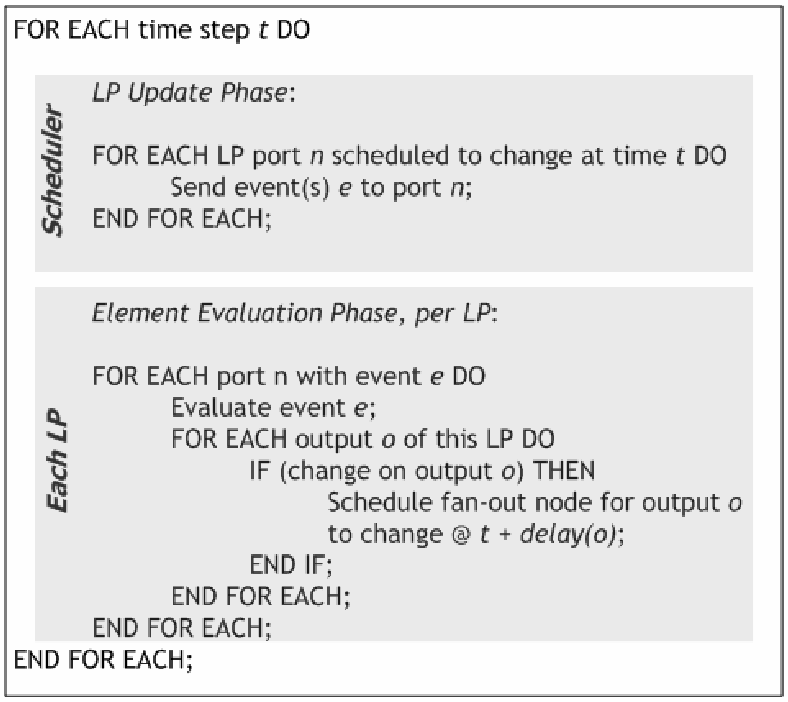


Figure 8 The sequential simulation (A) architecture and the asynchronous PDES (B) architecture

In the following two subsections, we go into more detail about the sequential DES and asynchronous PDES algorithms. These algorithms will be the foundation for the synchronization methodology used in the Chatoyant/ModelSim co-simulation interface.

3.1.1 Synchronous Simulation Methodology

In sequential DES, all LP nodes must process their events before the whole simulation can progress to the next time step (12). In the sequential methodology, there is a fixed difference between timestamps, Δ, such that every LP node will consume and produce events for some $k\Delta$ time later, where $k \geq 0$ (12, 16). Figure 9 shows the pseudo code for the sequential algorithm from (12). This code shows both the scheduler and LP process code for every time step.


Scheduler
Each LP

```
FOR EACH time step t DO
    LP Update Phase:

    FOR EACH LP port n scheduled to change at time t DO
        Send event(s) e to port n;
    END FOR EACH;

    Element Evaluation Phase, per LP:

    FOR EACH port n with event e DO
        Evaluate event e;
        FOR EACH output o of this LP DO
            IF (change on output o) THEN
                Schedule fan-out node for output o
                to change @ t + delay(o);
            END IF;
        END FOR EACH;
    END FOR EACH;
END FOR EACH;
```

Figure 9 Pseudo code for the synchronous PDES algorithm (12)

In the section marked as *Scheduler* in Figure 9, each LP is updated with the set of events bound for it. These events in turn activate their destination ports within the LP. In the *LP* segment, each LP will perform an evaluation for each event based upon this event and the current state of the LP, as seen in Figure 7. If there is a change generated by the output evaluation function, a new event is scheduled at some delay time later. This process will then repeat for every port of an LP that has a new event at time *t*. This process occurs for each LP at the given time step until all events for the current time step are consumed (12).

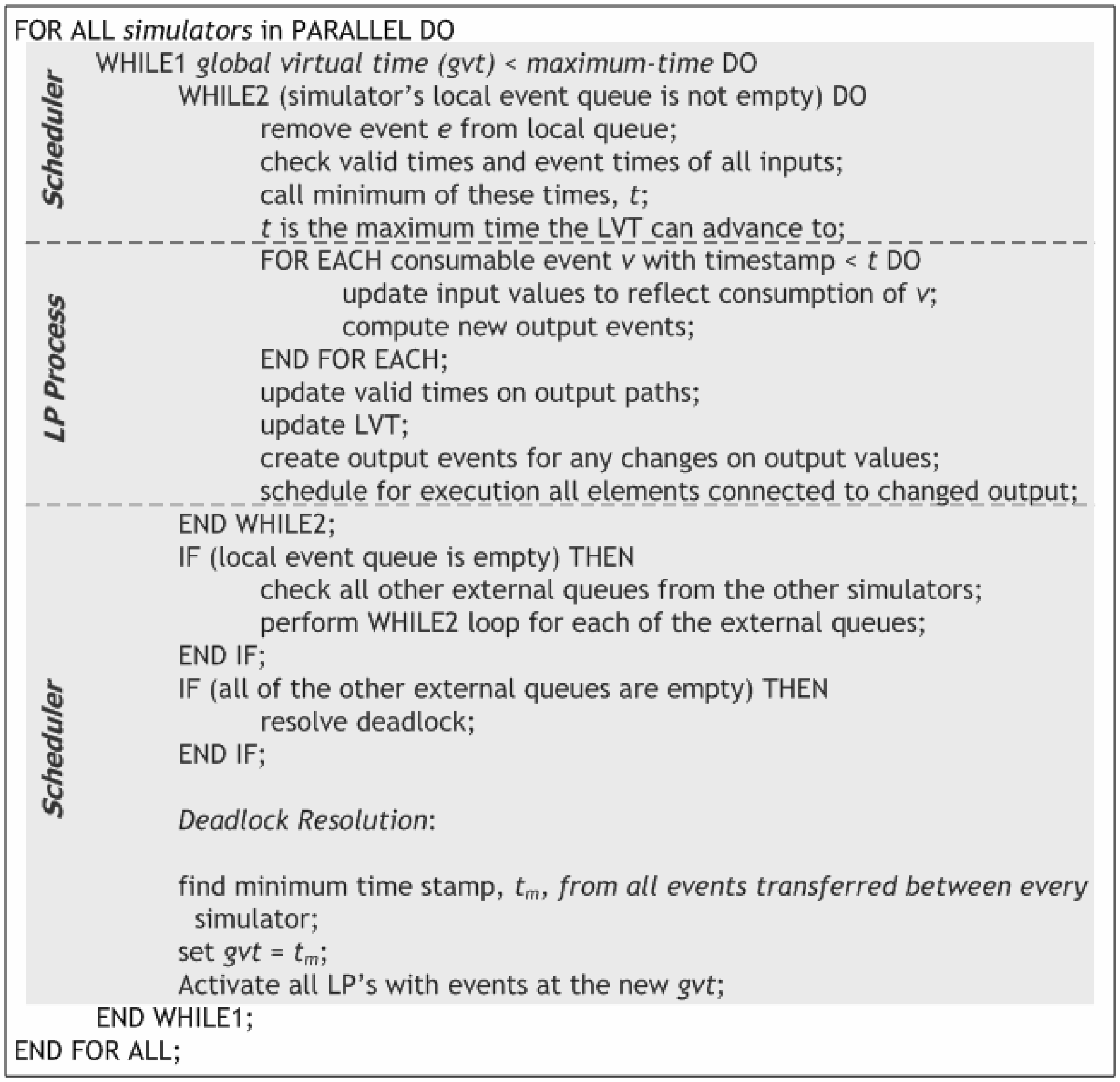
The algorithm seen in Figure 9 does have limitations. As discussed in (16), there are four factors that determine simulation overhead: frequency, duration, level of detail, and number of simulated events. The evaluation frequency is related to the inverse of the time-step, or $1/\Delta$. The duration of synchronization is defined by the time it takes each LP to execute and reach a quiescent state for a given time step. This factor is partially dependent on the maximum frequency as well as the length of time it takes an LP to perform a particular operation. The level of detail reflects the amount of processing potential at each LP. If an LP is capable of processing information faster, the simulation overhead will decrease. Likewise, the number of simulated events, which refers to the average quantity of events processed by each LP for a given time step, also reduces synchronization overhead if more events are processed in a short amount of time. In sequential DES, the finer the resolution of the time step frequency is the greater computational load is, placed on every LP. Since all LP's are executing sequentially in this type of simulation, the sum of the runtime for each LP times the number of events processed by each LP, proves to be a large performance inhibitor (12, 16).

To resolve the problems with sequential simulation, asynchronous parallel discrete event simulation is utilized (12). This branch of discrete event simulation provides a method for partitioning a design into smaller individual simulations that, in turn, simulate independently of one another, as seen in Figure 8b. This method of simulation has shown to have significant speed-ups depending on the system being simulated and type of synchronization used (12).

3.1.2 Asynchronous Simulation Methodology

To explain asynchronous simulation, the methodology, described by (12), is based on there being multiple simulators, each with their own scheduler and one or more LP's. Asynchronous PDES methods are divided by the two means of controlling their simulation, based on the concept that each simulator is permitted to progress to its respective local simulation time, otherwise known as its local virtual time, or LVT (11). Both approaches deal with a concept called temporal causality. Temporal causality is defined as the restriction that any event arriving from another node must not have a timestamp earlier than the destination node's LVT (11). The first type of synchronization, known as conservative, strictly adheres to maintaining causality. The second type of synchronization, known as optimistic, does allow causality to be broken with the ability to rollback to a known safe state in order to recover from the causality violation (11, 12 13). The different implementation approaches to both of these synchronization methods will now be discussed.

The conservative methods of synchronization, first developed by Chandy and Misra, as well as Bryant (11, 12), strictly prohibit violating causality. For any given time step, no LP is permitted to send or receive an event that is earlier than any other of the LVTs of the other LP nodes. To enforce this, there must be constant communication between the LP nodes. This method of simulation, however, is prone to deadlocks since there exists the possibility that every LP will not receive an event that permits itself and other simulators to progress to the next time stamp (12). To resolve this, deadlocks can be avoided by null message passing; deadlocks are recoverable by assuming that the event with the earliest timestamp is the safest event, or by means of lookahead (11, 12). The pseudo code in Figure 10 describes a general purpose conservative synchronization algorithm with deadlock resolution, according to (12).


Scheduler
LP Process
Scheduler

```
FOR ALL simulators in PARALLEL DO
    WHILE1 global virtual time (gvt) < maximum-time DO
        WHILE2 (simulator's local event queue is not empty) DO
            remove event e from local queue;
            check valid times and event times of all inputs;
            call minimum of these times, t;
            t is the maximum time the LVT can advance to;
            FOR EACH consumable event v with timestamp < t DO
                update input values to reflect consumption of v;
                compute new output events;
            END FOR EACH;
            update valid times on output paths;
            update LVT;
            create output events for any changes on output values;
            schedule for execution all elements connected to changed output;
        END WHILE2;
        IF (local event queue is empty) THEN
            check all other external queues from the other simulators;
            perform WHILE2 loop for each of the external queues;
        END IF;
        IF (all of the other external queues are empty) THEN
            resolve deadlock;
        END IF;

        Deadlock Resolution:

        find minimum time stamp, $t_m$, from all events transferred between every
          simulator;
        set $gvt = t_m$;
        Activate all LP's with events at the new gvt;
    END WHILE1;
END FOR ALL;
```

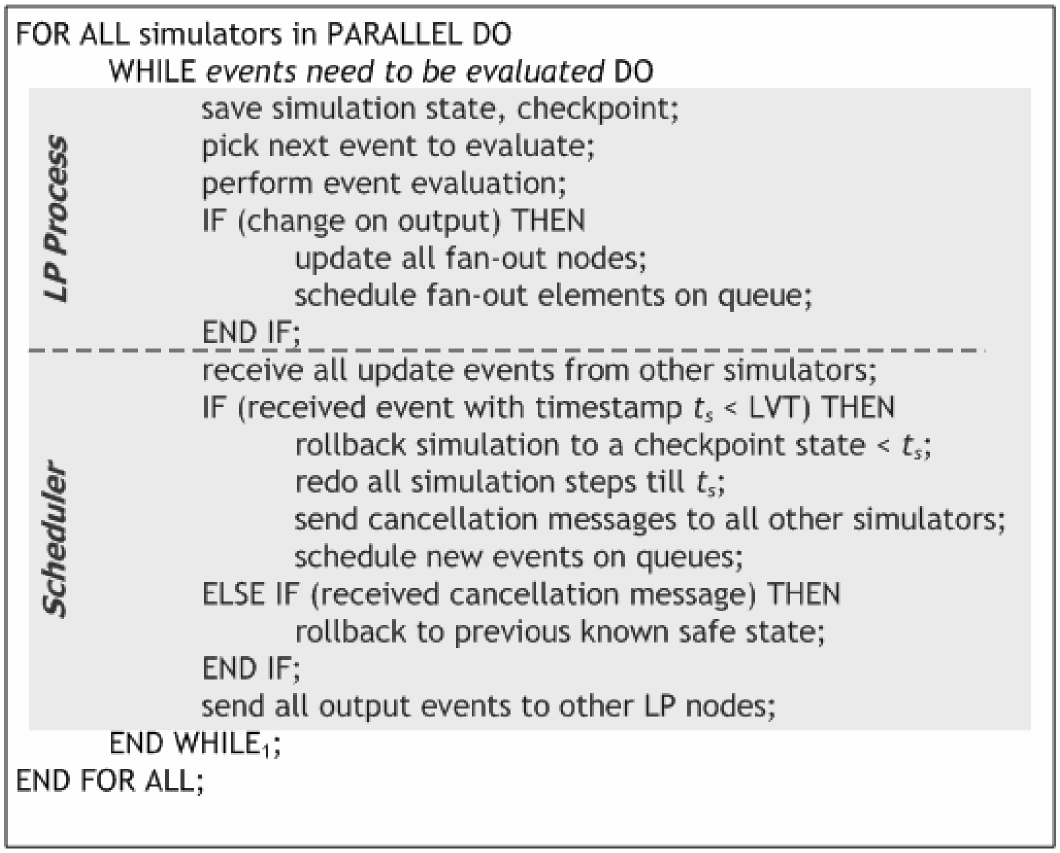
Figure 10 Asynchronous, conservative based PDES synchronization pseudo code (12)

In the pseudo code seen in Figure 10, the code is divided into three parts. The first part, related to the local scheduler of any of the simulators used for simulation (refer to Figure 8b), checks to see if the global virtual time, or *gvt*, is equal to the overall stop time of the simulation, denoted as *maximum-time*. Then for every event in the simulator's event queue, the event's timestamp is checked against the current LVT of the simulator. As soon as the minimum of these times is determined, the scheduler will then activate the LP and its computational section. Once all of the events at that time stamp have been processed and all output events have been issued to the scheduler, the third stage of the code is invoked. In this stage, all other incoming event queues are processed from the other simulators. The same event processing is performed until there are no more events. At this point, all event queues in each LP node are checked to determine if there is deadlock. If there is a deadlock, the simulators perform a deadlock recovery as seen in the segment of code labeled *Deadlock Resolution*. Once the deadlock is resolved, the simulation proceeds normally. There are several methods for handling deadlock. We investigate those methods now.

One of the first deadlock resolution methods is null message passing, as defined by Chandy/Misra and Bryant in (11, 12, 20). It states that an LP node will broadcast a null event with a safe timestamp as a promise that there will be no other events with timestamp t, such that $t > t_{null}$. This method is typically a solution that incurs high overhead since the communication system is loaded with additional messages (17). Deadlock recovery methods require mechanisms that first detect then resolves deadlocks as described in (21, 22). As noted in (11), deadlock recovery does reduce synchronization overhead significantly when compared to null message passing. However, deadlock recovery typically performs similarly to the synchronous system when approaching and resolving a deadlock situation, which has the potential for reducing the overall simulation performance (11).

The final conservative synchronization method is lookahead. This method provides both deadlock avoidance as well as limiting the need for null messages (11). Lookahead requires more implementation considerations than the preceding methods mentioned. One consideration includes how to perform the lookahead within the simulator. This means that there must be a method available to the simulator to be able to determine when the next time stamp is. Another consideration is what resources are available within the simulator to support creating a function if it does not exist. This is seen in Ptolemy's DE scheduler, where there is not a built-in method for performing the lookahead. Therefore, a look into the Ptolemy DE scheduler shows that there is an event queue object available. Having access to this event queue is important for implementing a lookahead function. The conservative approach to asynchronous PDES provides a noticeable runtime speedup over synchronous methods, as shown in (12).

To further speedup simulation beyond the conservative methods, the optimistic approach, as first proposed by Jefferson in (44), and later discussed by Nicol and Fujimoto in (13), provides a different methodology when considering the constraint of maintaining causality. Optimistic asynchronous simulation assumes that the probability of violating causality is small with respect to the total number of events that will be processed and produced (12, 13). With this assumption, any processing element is allowed to freely execute until it receives an event in the past with respect to its own LVT. When this violation occurs, the LP must stop and return to a previously known safe state, typically defined as the last known LVT which is earlier than the timestamp of the violating event. In order to accomplish this, the current state of any LP must be saved periodically so that if there is an event violating causality, the LP has a previously known safe state. Depending upon the number of elements within the LP and the complexity of the computation occurring within the LP, there is the potential for a large overhead both in terms of execution time as well as system resources (12). Also, each node must be capable of supporting the state saving ability (13). Figure 11 illustrates the pseudo code for a general purpose optimistic simulation, as described in (12).


LP Process
Scheduler

```
FOR ALL simulators in PARALLEL DO
        WHILE events need to be evaluated DO
                save simulation state, checkpoint;
                pick next event to evaluate;
                perform event evaluation;
                IF (change on output) THEN
                        update all fan-out nodes;
                        schedule fan-out elements on queue;
                END IF;
                receive all update events from other simulators;
                IF (received event with timestamp $t_s$ < LVT) THEN
                        rollback simulation to a checkpoint state < $t_s$;
                        redo all simulation steps till $t_s$;
                        send cancellation messages to all other simulators;
                        schedule new events on queues;
                ELSE IF (received cancellation message) THEN
                        rollback to previous known safe state;
                END IF;
                send all output events to other LP nodes;
        END WHILE$_1$;
END FOR ALL;
```

Figure 11 Asynchronous optimistic synchronization pseudo code reproduced from [12]

We can observe that the pseudo code described in Figure 11 is less complicated when compared to the pseudo code for the conservative method as seen in Figure 10. The pseudo code for the optimistic approach divides into two segments. First, the LP process will consume events and produce any output events. The second segment of code performs the scheduler functions. In the optimistic case, other events are received from other LP nodes and then checked against the LVT of the current node. If there is the timestamp of an event is less than the current node's LVT, then causality has been violated. At this point, the LP must rollback to the checkpoint that is less than t_s and simulate from that checkpoint to the timestamp of the violating event. Cancellation messages are sent from the current LP to other LP nodes to reverse any effects the rollback may have caused, and all new output events are scheduled. Regardless of whether there was a rollback, the LP performs a checkpoint and then proceeds to the next event and timestamp. This will repeat until there are no more events to be processed or a stop time has been reached.

Assuming there exists the resources to perform the checkpoint and restore for a rollback system, there are many methods discussed in (13) regarding how to determine when it is appropriate save the state of a simulation and how to efficiently perform a rollback. The optimistic asynchronous PDES methods have shown further speedups beyond the conservative methods, described in (12).

We can apply the concepts described within this section to the co-simulation interface between Chatoyant and ModelSim. Both the synchronous and asynchronous techniques provide a possible method of synchronization for the co-simulation interface. We will now further investigate these two synchronization techniques in the following chapter

3.2 USING PDES FOR THE CHATOYANT/MODELSIM CO-SIMULATION INTERFACE

We now apply the theory presented in the previous sections to the co-simulation interface between Chatoyant and ModelSim. To refresh: co-simulation is defined as one or more independent simulators executing in a cooperative fashion, each simulating a particular segment of a design (3, 5). As described in (3), one of the more critical components of co-simulation is synchronization. In this section, we investigate a cycle-accurate lock-step approach as well as using the conservative asynchronous PDES methodology as a dynamic synchronization approach.

3.2.1 Lock-Step Co-Simulation

A common synchronization technique used in co-simulation is referred to as lock-step synchronization (25). Lock-step synchronization means that every simulator that is part of a co-simulation will synchronize by a constant step size, which we will call δ. This step size, δ, which is defined as the "step" from synchronization to synchronization (25), is usually determined by a fixed simulation clock (5, 25). In Figure 12 we illustrate how *n* number of Discrete Event simulators will progress through time with using the lock-step co-simulation technique. This figure exemplifies how the step size restricts the simulators from allowing interim events to be available to the other simulators, if they occur on the signals that cross the co-simulation interface.

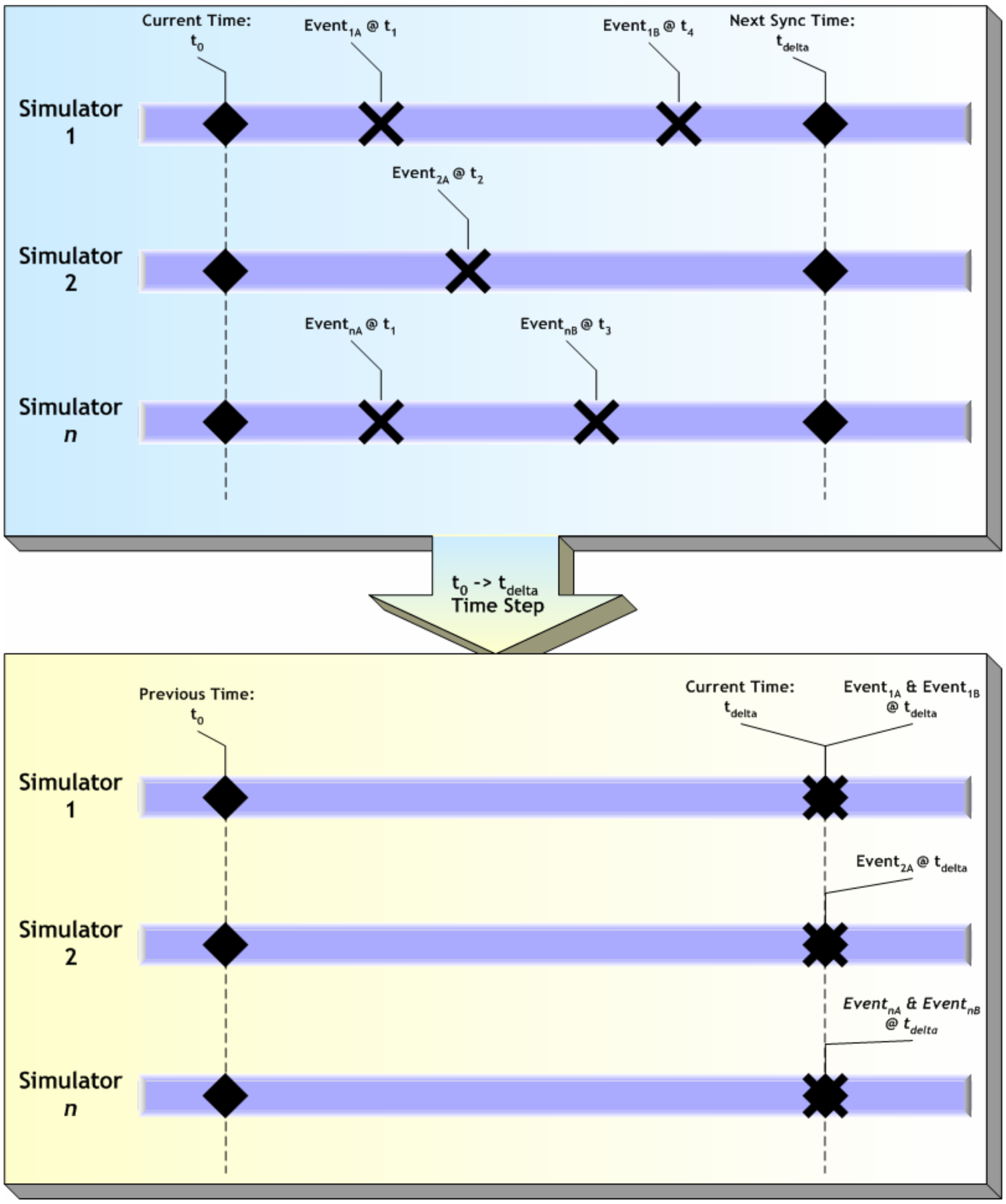


Figure 12 An example of a lock-step co-simulation

The diagram in Figure 12 depicts the event behavior between two synchronization cycles during the co-simulation. The top box shows the simulation state of simulators 1 through *n* at time t_0. At this point, each simulator has a set of events scheduled. Simulator 1 has an event, $Event_{1A}$, scheduled at t_1, as well as an event, $Event_{1B}$, at time t_4. Likewise, Simulator 2 has an event, $Event_{2A}$, scheduled at time t_2. Finally, Simulator n has an event, $Event_{nA}$, scheduled at time t_1 as well as an event, $Event_{nB}$, scheduled at time t_3. The next synchronization step is scheduled to occur at t_{delta}, which allows us to define the step size, δ, as $\delta = t_{delta} - t_0$. When the simulation reaches t_{delta}, all of the simulators synchronize with one another. This state of the interface is in part defined by the current values of the signals that traverse the interface between the different simulators (5). At the t_{delta} point in simulation time, the simulators only receive the current state of the signals that traverse the interface

As seen in the previous example, when performing lock-step co-simulation between discrete event simulators, the synchronization period has an impact on simulation accuracy. However, this period has the inverse effect on simulation runtime performance when considering the overhead associated with each simulator computing the next synchronization point (5, 6).

Let us now consider three different cases with respect to the magnitude of δ as it effects overhead and simulation accuracy. If we consider the case of δ being at least an order of magnitude greater than the resolution of the events being scheduled (for example $\delta = 10^{-9}$ versus a resolution $= 10^{-12}$), the associated synchronization overhead is reduced, and therefore the simulation performance is potentially increased (5, 6). However, the number of interim events visible across the interface is significantly reduced. If δ is equal to the resolution of the events being scheduled, there are an increased number of events that are visible at the synchronization point. However, this increases the synchronization overhead and thus impact the overall simulation performance. Finally, if the value of δ is significantly less than that of the simulation resolution (for example $\delta = 10^{-12}$ versus a resolution = 10^{-9}), every event will be visible at the synchronization points, but there is an even higher synchronization overhead imposed.

The lock-step method of synchronization does, as seen in the discussion above, have the pitfall of not always showing the entire event traffic. However, for the perspective that we are interested in with the Chatoyant/ModelSim co-simulation interface, we would like to maintain as much event visibility across the co-simulation interface as possible without incurring a large overhead. To address this desire, we will now look at applying the asynchronous PDES methods to co-simulation.

3.2.2 Dynamic Co-Simulation

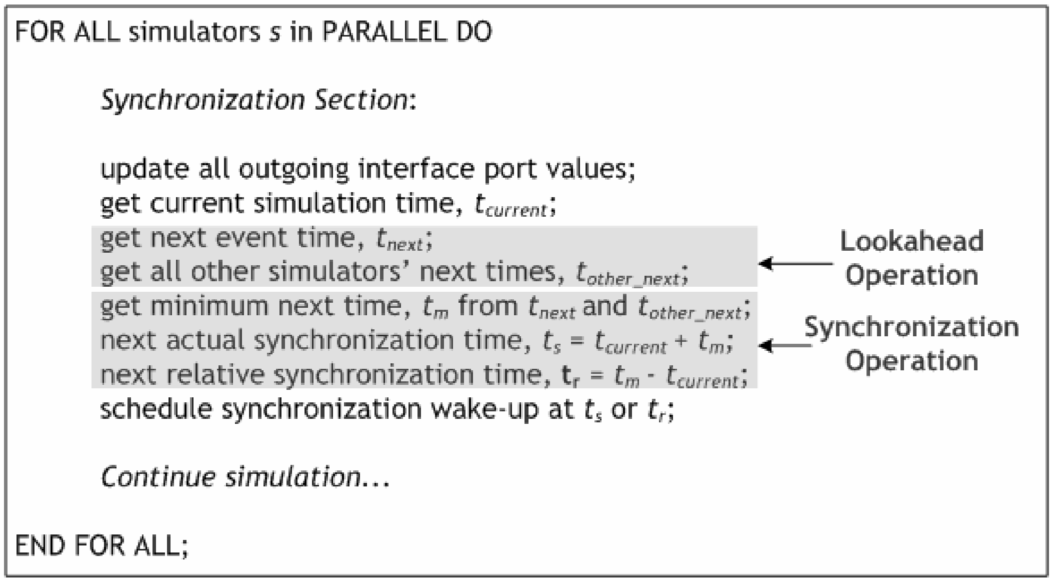
As we have stated, one of the goals of the co-simulation interface is to provide an event accurate co-simulation. Using the cycle-accurate co-simulation methodology, discussed in the previous section, leads to some events not being seen across the interface. These events can represent critical occurrences that can directly impact how a system behaves. Therefore it is necessary to have in place an alternative to the lock-step approach for synchronization in a co-simulation environment. The asynchronous approach seen in PDES becomes an answer for both the runtime issues as well as accuracy deficiencies.

When considering the application of asynchronous PDES to co-simulation, we must evaluate both the conservative and optimistic methods of synchronization, described in Section 3.1.2. One issue we must consider is that co-simulation is typically performed on one workstation, especially in the case of the Chatoyant/ModelSim interface (3, 25). This places a constraint on the CPU power and system memory that is available to the simulator programs. Therefore, when we consider using an optimistic approach, the need for period checkpoints, rollback overhead, and support for these two functions becomes critical. To perform a checkpoint, even if performed in an efficient way as proposed by (26), consumes memory and perhaps disk space, as seen in (30). A rollback also requires loading a checkpoint from disk back into memory and potentially restarting the simulation from that point, as seen in ModelSim (30).

When considering the optimistic approach, we must also make the assumption made that all of the simulators being used have the ability to checkpoint and rollback. In the case of the Chatoyant/ModelSim co-simulation interface, ModelSim has mechanisms to perform the checkpoint and restore operations, however, Chatoyant does not. To implement this feature in Chatoyant is difficult and at best a time consuming process since we must add this ability to all Chatoyant stars as well as directly modifying the discrete event scheduler in Ptolemy. For these reasons we focus of on using a conservative approach for synchronization.

The conservative synchronization provides a noticeable speedup without the overhead of optimistic methodologies (12). To implement the conservative synchronization methods, we need to have a notion as to every simulator's place in time as well as a method for avoiding deadlock (23). For each simulator to have insight into the others' time state, a special communication packet or a shared memory location is used. To avoid deadlock, we must choose ether a null message scheme, a deadlock recovery scheme, or a lookahead method.

The critical issue concerning the use of conservative PDES is deadlock resolution. As stated in (17), the null message scheme used to solve the deadlock problem can be very costly since it utilizes much of avaialable communication bandwidth. Deadlock recovery systems require extra resources to allow an undo, as seen in (12, 15). Lookahead provides a powerful mechanism, given that it is available or easily implemented in the simulator being used (16). Utilizing the predictive barrier scheduling scheme proposed in (16), we can take advantage of searching the event queue of each simulator to determine when the next synchronization should be scheduled. We define the lookahead operation for each simulator in the pseudo code in Figure 13. This pseudo code is adapted from the algorithm presented in [16] as it applies to the Chatoyant/ModelSim co-simulation interface.



```
FOR ALL simulators s in PARALLEL DO

        Synchronization Section:

        update all outgoing interface port values;
        get current simulation time, $t_{current}$;
        get next event time, $t_{next}$;
        get all other simulators' next times, $t_{other\_next}$;
        get minimum next time, $t_m$ from $t_{next}$ and $t_{other\_next}$;
        next actual synchronization time, $t_s = t_{current} + t_m$;
        next relative synchronization time, $t_r = t_m - t_{current}$;
        schedule synchronization wake-up at $t_s$ or $t_r$;

        Continue simulation...

END FOR ALL;
```

Figure 13 Dynamic synchronization pseudo code adapted from [16]

As we see in Figure 13, there is a lookahead operation and a synchronization operation performed. First, all of the simulators update the port values that serve as outputs to the other simulator and each simulator posts its current time to the other simulators. Secondly, both of the simulators determine their next event times, based upon a lookup within their respective event queues. Third, each simulator posts its next event time to the other simulators [12]. When each simulator has all of the next event times, they will all determine the minimum of these times. This minimum time defines the safe time interval that all simulators can execute before synchronization is performed [16]. The synchronization operation performs two calculations, depending on how a simulator schedules events. If a simulator requires that events (we define synchronization wake-up as an event) be time stamped with the actual time of occurrence, the t_s value is used. This is seen in the case of Chatoyant and Ptolemy [7, 9]. However, if the simulator requires that events be time stamped with a relative time of occurrence, as seen in ModelSim [30, 31], the t_r value is used. With the current wake-up time determined, this value is scheduled and the simulation continues.

With the current simulation time and the next event time, synchronization is reduced to a series of integer comparisons in every simulator without requiring complicated calculations or an over utilization of I/O operations. As long as the lookahead functionality is available, as well as the shared memory discussed earlier, the conservative co-simulation approach provides an efficient and easily implemented interface. In Figure 14, we illustrate an example of the dynamic synchronization for *n* simulators.

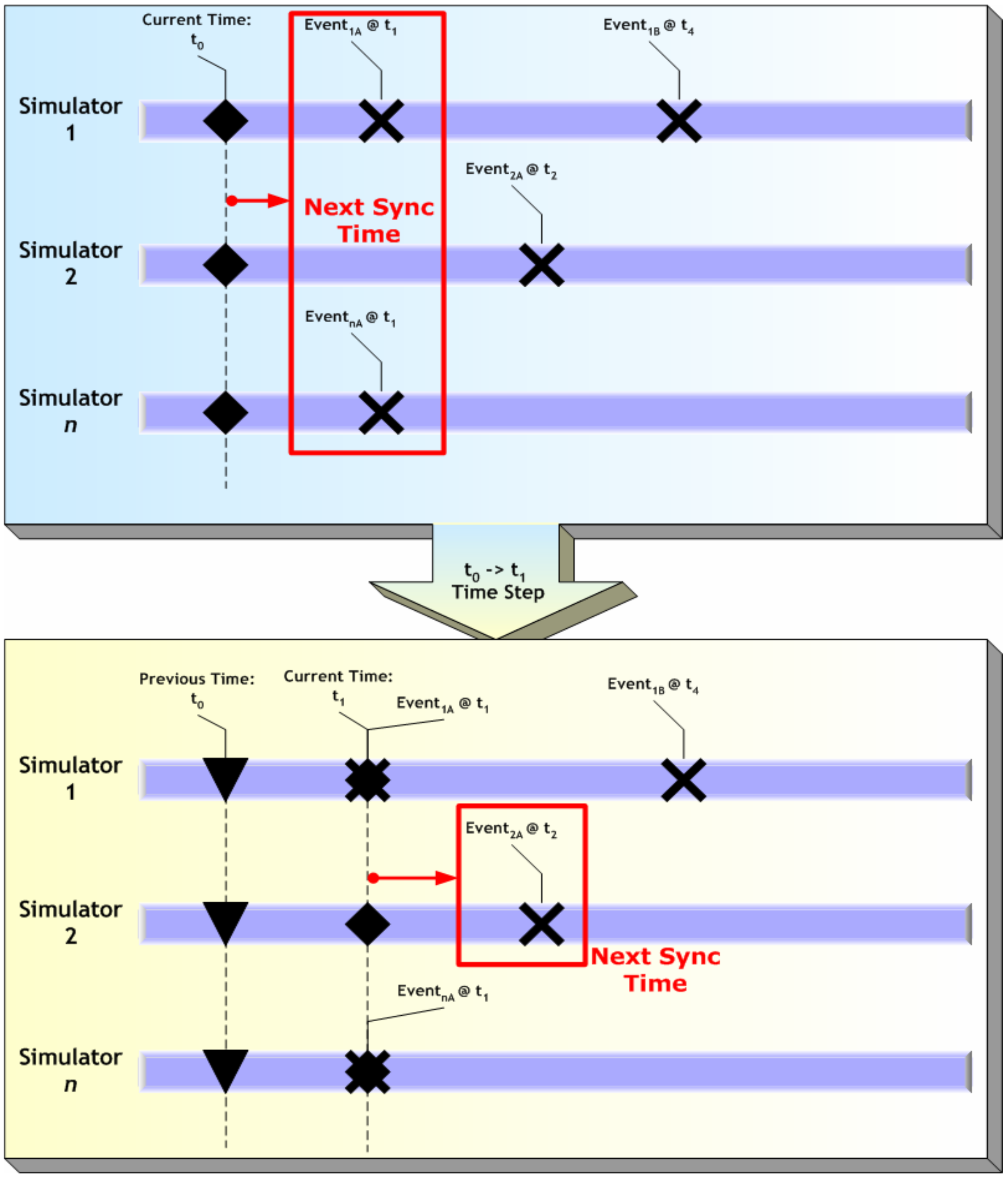


Figure 14 Our example of dynamic synchronization

Similar to the example seen in Figure 12 for the lock-step method of synchronization, Figure 14 shows how we perceive the dynamic algorithm operates. Once again we show n simulators stepping through time, and all start at t_0. Simulator 1 has events scheduled at t_1 ($Event_{1A}$) and t_4 ($Event_{1B}$). Simulator 2 has one event, $Event_{2A}$, scheduled at t_2. Finally, Simulator n, has one event, $Event_{nA}$, scheduled at t_1. Using the dynamic synchronization method, each simulator looks at their event queues and determines the next event times. For Simulator 1, this time is t_1, Simulator 2 determines t_2 to be its next event time, and Simulator n determines its next time to be t_1. Next, all of the simulators post their next time values to one another and all calculate the next synchronization time. Time is set equal to $t_{1,}$ which is the earliest time. Each simulator schedules a synchronization wake-up event at t_1 and proceeds with the simulation. When the simulators synchronize at t_1, they perform the lookahead and posting operation again to determine that this time, $t_{2,}$ is the valid synchronization time. This continues until the end of the simulation.

With this method of synchronization, deadlock is avoided since no simulator waits for an event that is known not to violate causality. With the issue of synchronization covered, we now shift our focus to how event data is translated from one physical domain, such as digital electronics (i.e., multi-valued logic), to another physical domain, such as analog electronics (i.e., voltages).

3.3 DIGITAL/ANALOG SIGNAL TRANSFORMATION

With a basic understanding of temporal synchronization of discrete event simulation, another consideration of co-simulation is that of communicating between the different physical domains represented by each simulator. In this context, a physical domain is defined as the set of common characteristics of a signal groups that exist in a particular segment of a design. We define the simulation of multiple physical domains as mixed-signal simulation, as seen in (1, 2, 19). Since we have been referring to MOEM systems, the possible physical domains are optical, mechanical, electrical, and now digital. In the case of a co-simulation environment, the interface between the multiple simulators must handle the conversion between these physical domains and convert from one signal type to another. Even in the case of a unified language set such as VHDL-AMS, there is still the consideration of how signals are translated between the different physical domains (10). For example, a VHDL-AMS component defines how it converts a logic literal into a voltage, governed by a set of physical rules.

In the co-simulation interface between Chatoyant and ModelSim, the interface has to convert between analog and digital signal types. The conversion between analog and digital signal values becomes a technology specific concern, meaning that how digital values translate to voltages and vice versa is dependent on the properties of an integrated circuit (e.g., 0.13 micron low voltage CMOS integrated circuit). Different voltage levels correspond to different logic values, depending on how the integrated circuit reacts to certain voltages.

To accurately model the translation between digital and analog signals, we make use of the Voltage-Transfer Characteristic, or VTC, as provided in [26]. The VTC plots the output voltage as a function of the input voltage for a given gate. This plot provides the necessary information for mapping logic values to voltages, and vice versa [26]. Figure 15 illustrates an example VTC and the corresponding relationship to logic values.

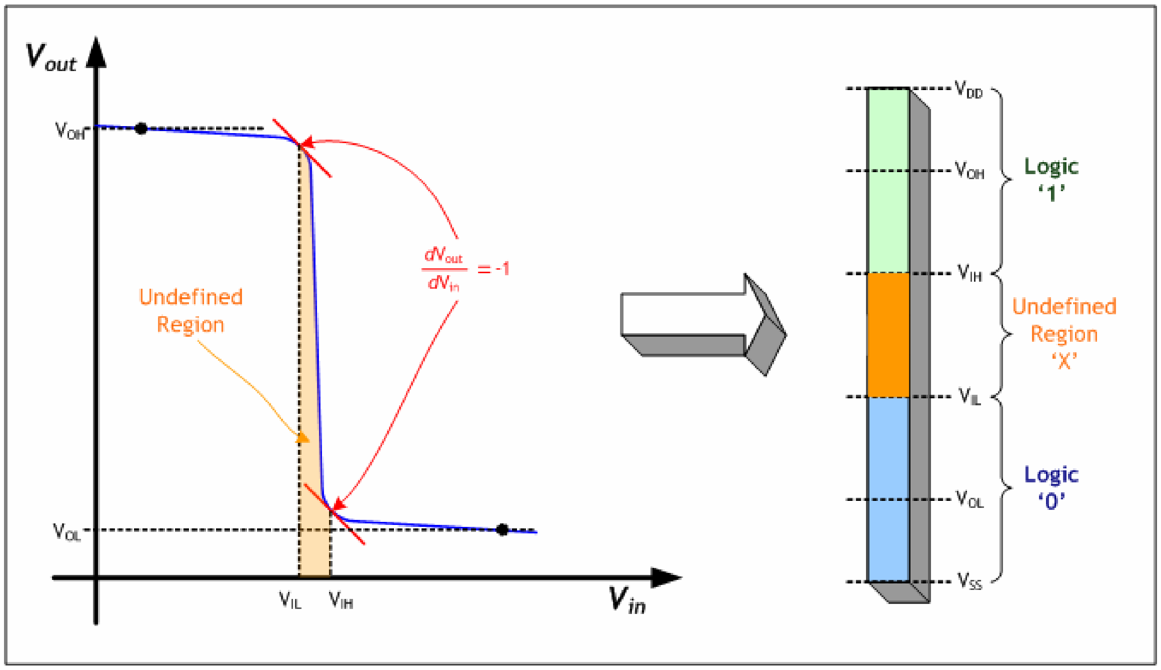


Figure 15 Sample voltage threshold graph for TTL and CMOS technologies adapted from [26]

The graph in the left side of Figure 15 depicts a VTC curve. There are four primary points of interest concerning this graph. The point marked as V_{OH} denotes the lower limit of a gate's output voltage producing an equivalent logic '1', as designed and manufactured for a given power supply voltage. Likewise, the point marked at V_{OL} denotes the upper limit of a gate's output voltage producing an equivalent logic '0'. There are also two points on the curve where its derivative, signified by the equation dV_{out}/dV_{in}, is equal to -1 (26). These points represent the limits where the upper limit of the input voltage, V_{IL}, is equal to a logic '0' and the lower limit of the input voltage, V_{IH}, is equal to a logic '1' (26). The area under the VTC curve, bounded within V_{IL} to $V_{IH,}$ is considered to be undefined and marks the region of operation where proper operation for a steady-state signal cannot be guaranteed (26). The right side of Figure 15 provides a chart that defines the different logic regions in terms of the voltage boundaries determined by the VTC curve. This mapping is the method used to define the voltage/logic conversions within the Chatoyant/ModelSim co-simulation interface. The current interface requires that the user provides the V_{IH} and V_{IL} limits as well as the highest and lowest allowable voltages for each port that crosses the interface.

There is also a way for taking in account for drive strength when converting between analog and digital signals. The IEEE Verilog and VHDL standards both contain types for logic '1', '0', high-impedance, and unknown ('X') (27). VHDL provides a more detailed list of types by providing for weak (weak in the sense that the driving circuitry behind the logic can be over powered a stronger one (31)) one's and zero's and unknown's (27, 31). This is typically performed by taking in account the output impedance of the signal driver as well as what the device receiving a signal considers a strong or weak driver. Currently, the Chatoyant/ModelSim interface assumes all signals are strong. Another and there is no support for high impedance, 'Z'. However, the Chatoyant message class has parameter fields which provide one of the mechanisms for performing the drive strength determination. The other mechanism needed is a list of parameters within the co-simulation interface star in Chatoyant for calculating whether a port drives a strong or a weak value onto a signal.

We provide an example of a voltage/logic mapping in Table 1. In this table, we provide the voltage and logic values determined by the VTC for a CMOS Inverter, seen in (26). In VHDL, logic values are defined by a standard known as the 9 level Multi-Value Logic, or MVL-9 (27). The table shows the voltage level for the different regions of the inverter, the MVL-9 definition, and the corresponding MVL-9 character found in VHDL.

Table 1 Voltage to MVL-9 mapping for a generic 5V CMOS technology (voltages provided in (26))

Region	Voltage Level	MVL-9 Value	MVL-9 Character
Low Output Voltage	0.0V	Strong/Weak Low	‘0’ or ‘L’
Low Input Voltage	2.06V	Strong/Weak Low	‘0’ or ‘L’
Threshold	2.51V	Undefined	‘X’
High Input Voltage	2.92V	Strong/Weak High	‘1’ or ‘H’
High Output Voltage	5.0V	Strong/Weak High	‘1’ or ‘H’

With voltage-logic mapping defined, we can apply the same theory to any technology and logic value system that would be found in the co-simulation interface. The conversion between voltage and logic and vice versa will later be implemented as a look up table within Chatoyant, as seen in Chapter 4. We will now investigate the topic of performance measurement for the co-simulation interface between Chatoyant and ModelSim.

3.4 CO-SIMULATION PERFORMANCE ANALYSIS

With the synchronization and event communication methodology understood, we now consider how to measure the performance of the co-simulation interface between Chatoyant and ModelSim. We define a specific metric for measuring runtime performance. This metric is the number of events per second that cross the co-simulation interface. We also use code profiles from both simulators to determine where the largest percentage of the execution time is being spent. This is a tool that pinpoints bottlenecks within the interface code.

The metric used to measure performance for the Chatoyant/ModelSim co-simulation interface is that of events per second, which simply indicates how many events pass through the interface per second of wall-clock execution time. This measure varies based on input stimulus on both sides of the interface as well as the type of signals that pass through the interface.

The other performance measurement we use is a code profile on both the ModelSim and Chatoyant interface code. For Chatoyant, this metric is further divided into interface-related execution, event processing, and miscellaneous processing. This provides us with valuable insight into the performance impact. The ModelSim code profile is divided into two segments. The first segment profiles the synchronization and other functions found within the interface code. The second segment represents the remainder of the ModelSim execution, which encapsulates event processing and computation processes. Using these profiles, we are to evaluate the cost for processing an event in the interface and how this impacts the scalability of the co-simulation interface for larger systems.

In the following chapter, we describe how the theory discussed in this chapter can be applied to the Chatoyant/ModelSim co-simulation interface. Later, it is shown that co-simulation provides a solution for performing efficient system-level design exploration in the Chatoyant toolset.

4.0 CO-SIMULATION ENVIRONMENT IMPLEMENTATION

In this chapter, the actual co-simulation environment is discussed. As noted in Chapter 1, the co-simulation is between Mentor Graphic's subsidiary Model Technology's ModelSim, a mixed language simulator and the University of Pittsburgh's Chatoyant. As discussed in Chapter 2, Chatoyant is a simulation environment built upon the University of California at Berkeley's Ptolemy multi-domain simulation framework. Since neither Chatoyant or Ptolemy support direct HDL language simulation, a need was presented for a co-simulation environment. The following chapter discusses the architectural environment, the applications of parallel discrete event simulation, the considerations for the analog/digital domain crossovers, and finally a system specific interface generator tool that automates the creation of all interface related components as well as code generated for both ModelSim and Chatoyant.

4.1 GENERAL CO-SIMULATION ARCHITECTURE

Based on the background provided by Chapters 2 and 3, the co-simulation architecture for executing Chatoyant and ModelSim is defined by two major sub-components. The first of these is the communication system the two simulators use to synchronize and exchange event information. The second is the method of synchronization used to maintain temporal causality. Once these items are identified a general design can be derived and implemented.

In order to create a communication subsystem in the interface, it is necessary to consider a link that is as fast as possible. Since overhead is also a consideration, questions concerning message content, event information, and even channel mechanisms (pathway determination) are a focal point for discussion. Different implementation possibilities must be researched to find the best. Finally, thought must be given as to the structure of the event that crosses the interface

For communication architectures, many possibilities were examined. First a bidirectional message pipe system was considered. This would inherently provide a method of synchronization since both sides of the pipe (multidirectional) would have to be ready to receive and likewise send in order to continue execution. This does, however, present problems in terms of bandwidth utilization as well as having an implicit deadlock potential. By carefully implementing the interface, these potential problems are avoided and we can fully utilize the features provided by the pipes. This caution is exercised in the implementation of the interface, discussed later in this chapter.

The next possibility is shared memory. This method provides both simulators with a general repository of the ports that cross the interface as well as a place to keep the other simulator informed for time synchronization. For this methodology to be utilized only one stipulation must be met: both simulators must be synchronized in terms of the execution of the interface code running in both simulators. Without this, there is a possibility of a simulator receiving stale time and event data while the other simulator has already proceeded. Utilizing a semaphore or a blocking message FIFO readily solves this issue. Therefore, there are no event or time coherency problems.

With the communication sub-system defined, the more challenging aspect the co-simulation interface is time synchronization. Making use of the information provided in the previous chapter, two options are viable, lock-step and dynamic synchronization. Lock-step is the easier to implement since the synchronization is consistent and requires no major effort to determine when the next time to synchronize is. The dynamic method is more difficult to implement since it requires access to certain runtime data. It also performs comparisons and calculations to determine the next step time. Both of these methods are implemented for comparison purposes since the lock-step method is useful for cycle-accurate analysis and the dynamic method is useful for a more detailed simulation. The following sections provide a detailed explanation of the implementation of the co-simulation interface.

4.2 IMPLEMENTATION

The co-simulation environment was implemented in the Linux operating system. As a result, the architecture described above was partly created utilizing POSIX IPC (Inter-Process Communication) mechanisms such as shared memory and named pipes. Figure 16 shows the general block diagram of the co-simulation architecture with respect to its Linux-based implementation. This diagram depicts the underlying communication structure along with the shared memory arrangement and the simulators.

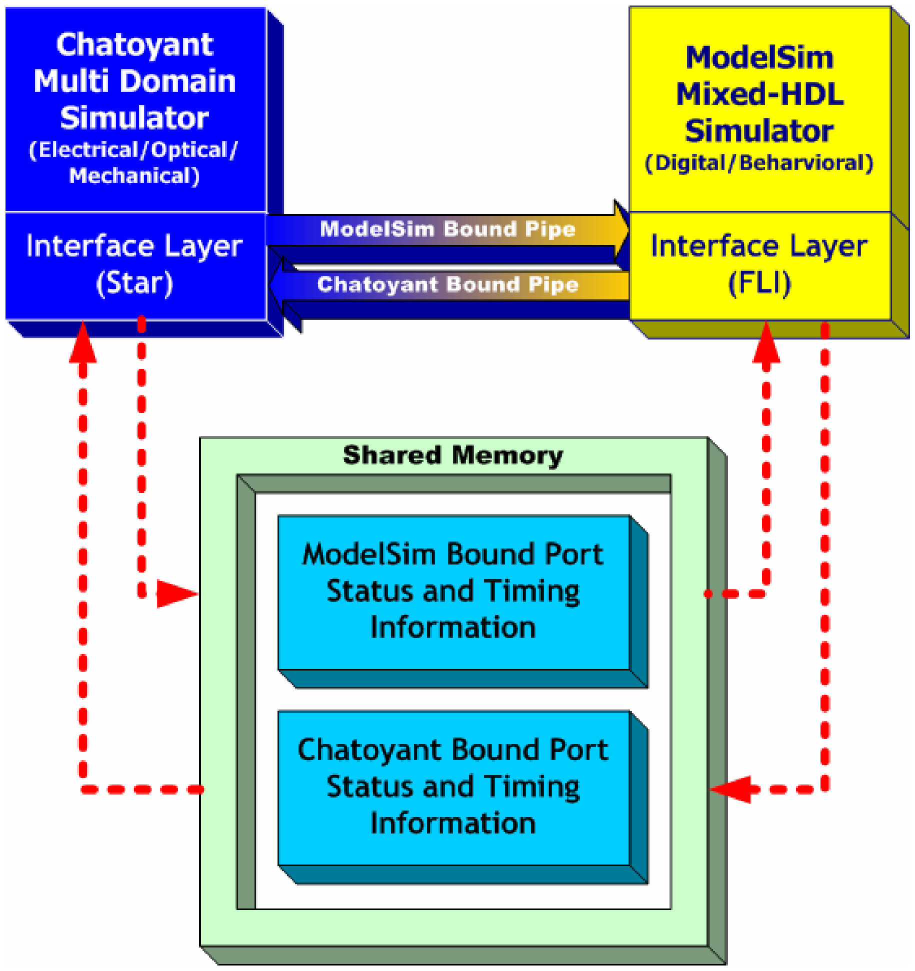
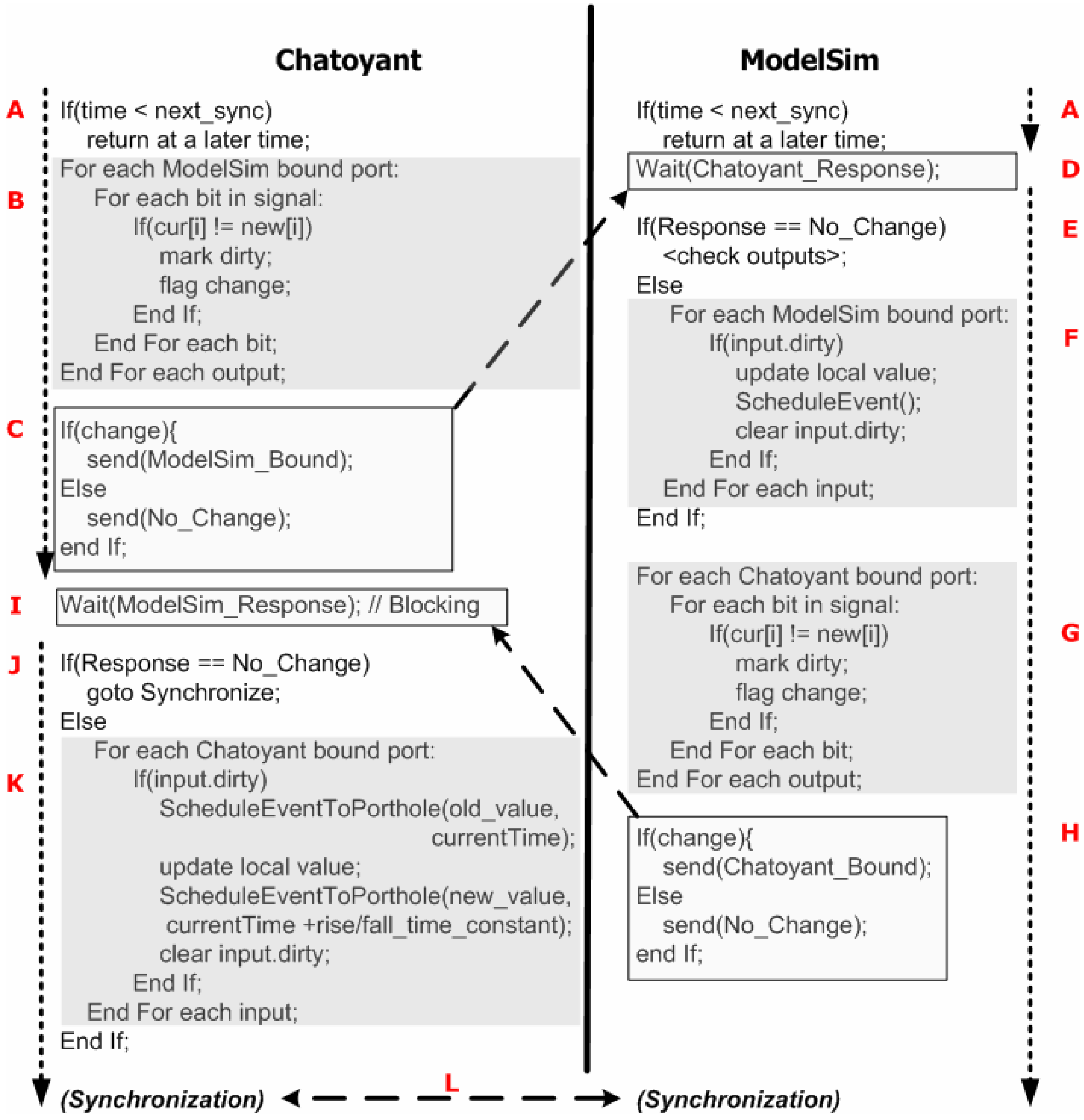

Chatoyant Multi Domain Simulator
(Electrical/Optical/ Mechanical)
Interface Layer (Star)
ModelSim Mixed-HDL Simulator
(Digital/Beharvioral)
Interface Layer (FLI)
ModelSim Bound Pipe
Chatoyant Bound Pipe
Shared Memory
ModelSim Bound Port Status and Timing Information
Chatoyant Bound Port Status and Timing Information

Figure 16 ModelSim/Chatoyant co-simulation architecture based on a Linux implementation

A large portion of the code body for the Chatoyant interface star and ModelSim FLI interface code is the same for both types of synchronization. This segment of code is responsible for the updating of every port that crosses the interface. Though this code is the same for each synchronization method, it is unique to a system meaning that it is not reusable with any other systems. This limitation occurs because every port is checked and updated and this is implemented by referencing variables that contain the port's name. The segment of code responsible for event communication is illustrated in Figure 17.



```
Chatoyant

A  If(time < next_sync)
     return at a later time;
B  For each ModelSim bound port:
       For each bit in signal:
           If(cur[i] != new[i])
               mark dirty;
               flag change;
           End If;
       End For each bit;
   End For each output;
C  If(change){
     send(ModelSim_Bound);
   Else
     send(No_Change);
   end If;
I  Wait(ModelSim_Response); // Blocking
J  If(Response == No_Change)
     goto Synchronize;
   Else
K      For each Chatoyant bound port:
           If(input.dirty)
               ScheduleEventToPorthole(old_value,
                                        currentTime);
               update local value;
               ScheduleEventToPorthole(new_value,
                currentTime +rise/fall_time_constant);
               clear input.dirty;
           End If;
       End For each input;
   End If;
   (Synchronization)
```

```
ModelSim

A  If(time < next_sync)
     return at a later time;
D  Wait(Chatoyant_Response);
E  If(Response == No_Change)
     <check outputs>;
   Else
F      For each ModelSim bound port:
           If(input.dirty)
               update local value;
               ScheduleEvent();
               clear input.dirty;
           End If;
       End For each input;
   End If;
G  For each Chatoyant bound port:
       For each bit in signal:
           If(cur[i] != new[i])
               mark dirty;
               flag change;
           End If;
       End For each bit;
   End For each output;
H  If(change){
     send(Chatoyant_Bound);
   Else
     send(No_Change);
   end If;
   (Synchronization)
```

Figure 17 Synchronization independent interface pseudo code for both ModelSim and Chatoyant

As seen in Figure 17, there are several sub-segments to the operation of this code, as it executes for every time step (either periodic in lock-step mode or at the next synchronization/event occurrence for the dynamic mode). Each major segment is marked by a letter for explanation purposes. In segment A, both simulators will determine if it is time to synchronize. If the current time in each simulator is less then the next synchronization time, each simulator will exit their respective synchronization functions and continue on with their simulations. If the current time is a valid synchronization point then the code in Chatoyant will proceed to segment B and ModelSim will proceed to wait for Chatoyant to post a response through the pipe, seen in segment D. Within segment B, Chatoyant will check every port bound for ModelSim to determine if there is any new data. If there is, it makes the appropriate update along with setting the dirty flag in the shared memory and sets a flag that there has been an update. After checking all of the ModelSim bound ports, Chatoyant moves to segment C, where, based on the update flag's status, it will either post a ModelSim bound change message or a null message if there were no changes to report. Chatoyant now moves to segment I where it waits for a message to be posted in the pipe coming from ModelSim.

At this time, ModelSim proceeds from segment D to segment E. If the message from Chatoyant is a null, ModelSim will proceed to segment G. Otherwise, ModelSim proceeds to segment F where it scans the dirty flags in the shared memory for all ports coming from Chatoyant. If there is a dirty flag set to true, ModelSim calls the force function from the FLI and drives the new value onto that signal. ModelSim now enters segment G where it checks all for any state change on ports bound for Chatoyant. As in Chatoyant's segment B, ModelSim determines if any signal has changed and if there is a change, posts the new value and sets the dirty flag in the shared memory. ModelSim then writes either a Chatoyant bound port value change or a null, seen in segment H. ModelSim now proceeds to its synchronization routine.

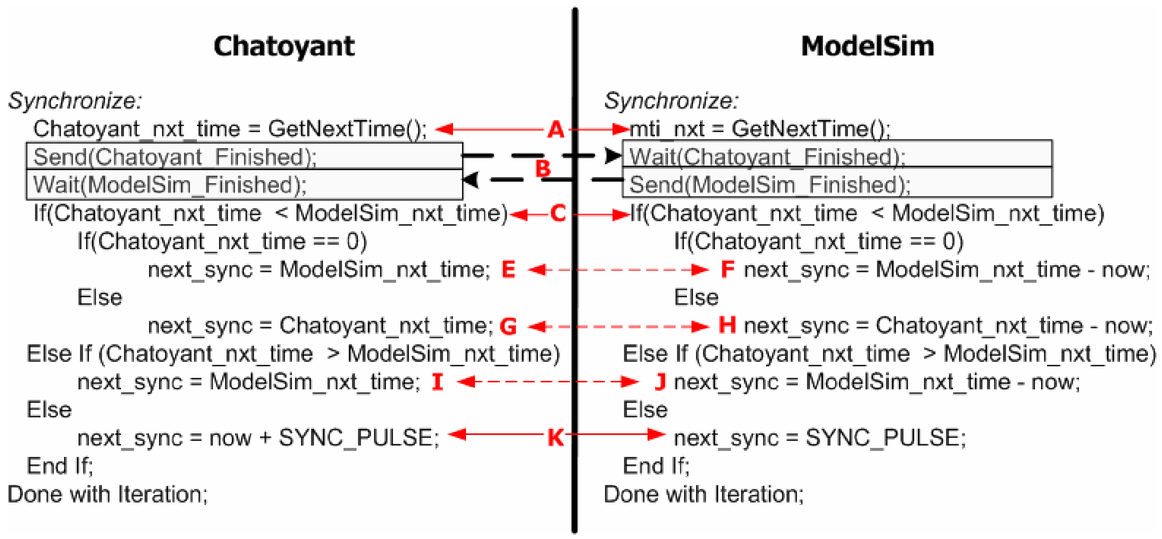
When Chatoyant receives the message from ModelSim via the pipe coming from ModelSim, Chatoyant performs the same operation in segment J as ModelSim did in segment F.

If there were no changes posted by ModelSim, Chatoyant proceeds directly to its synchronization routine. Otherwise, Chatoyant checks the dirty bit of every port coming from ModelSim, from the shared memory. As seen in segment K, any port that has a dirty flag set true is updated by Chatoyant converting the logic value to the proper voltage, posting the old value to porthole for that signal at the current time, and finally posting the new value for the signal at the current time plus the rise/fall time constant. After segment K, Chatoyant moves to its synchronization segment where both Chatoyant and ModelSim perform the time synchronization, represented by segment L.

As seen in the pseudo code in Figure 17, the checking and updating of port information is staggered. This means that Chatoyant and ModelSim do not simultaneously check the ports bound to the other interface before proceeding. This process is parallelizable in the sense that all outbound ports are checked before either simulator synchronizes with the other. For systems that contain large quantities of ports between Chatoyant and ModelSim, this presents a potential bottleneck. The reason the method shown in Figure 17 is used is because we want to support the ability to cause ModelSim to iterate through a delta cycle. Though this feature is not implemented at this time, it is available for future enhancements. If we find that there is a demand to making the code more efficient we can make the adjustment easily. We now proceed to discuss the two different synchronization methods as they are implemented in the Chatoyant and ModelSim interface code.

As for the synchronization segments, there are two different implementations to consider. The lock-step system is the simpler of the synchronization algorithms to implement. This method sets the next synchronization time to be the current time plus the constant synchronization period (defined by the user), called *SYNC_PULSE*. This constant is set in the Chatoyant star parameters and by means of a generic within the VHDL container for the FLI code loaded in ModelSim.

The dynamic synchronization approach, since it utilizes a lookahead algorithm, requires two steps. First, the synchronization process requires each respective simulator to get their respective next event time and post it to the shared memory. Once this information is posted, the simulators can determine which next event time is the least, and thus, set their next synchronization time to this value. This is illustrated through the pseudo code in Figure 18.



```
Chatoyant

Synchronize:
  Chatoyant_nxt_time = GetNextTime();
  Send(Chatoyant_Finished);
  Wait(ModelSim_Finished);
  If(Chatoyant_nxt_time  < ModelSim_nxt_time)
      If(Chatoyant_nxt_time == 0)
          next_sync = ModelSim_nxt_time;
      Else
          next_sync = Chatoyant_nxt_time;
  Else If (Chatoyant_nxt_time  > ModelSim_nxt_time)
      next_sync = ModelSim_nxt_time;
  Else
      next_sync = now + SYNC_PULSE;
  End If;
Done with Iteration;
```

```
ModelSim

Synchronize:
  mti_nxt = GetNextTime();
  Wait(Chatoyant_Finished);
  Send(ModelSim_Finished);
  If(Chatoyant_nxt_time  < ModelSim_nxt_time)
      If(Chatoyant_nxt_time == 0)
          next_sync = ModelSim_nxt_time - now;
      Else
          next_sync = Chatoyant_nxt_time - now;
  Else If (Chatoyant_nxt_time  > ModelSim_nxt_time)
      next_sync = ModelSim_nxt_time - now;
  Else
      next_sync = SYNC_PULSE;
  End If;
Done with Iteration;
```

Figure 18 The dynamic synchronization pseudo code

The pseudo code in Figure 18 is annotated with letters signifying each important segment. One important point is that the code in Figure 18 is appended to the code in Figure 17, if we are running in dynamic synchronization mode. In segment A, both simulators perform the lookahead function and get their respective next event times. They then post this time along with their current times to the shared memory and post that this operation is complete, seen in segment B. The simulators now determine when the next synchronization point is based on all of the current and next times. It is important that each simulator perform the same logic so that they both derive the same answer.

In segment C, both simulators first test to determine if Chatoyant's next event time is less than ModelSim's next event time. If it is, we must check to see if Chatoyant's event queue is empty. In the case where ModelSim is the primary source of events from the viewpoint of Chatoyant, it is possible that Chatoyant has no possible future events, and thus we are required to take further action. To handle this case, Chatoyant, in segment E, and ModelSim, in segment F, will use ModelSim's next event time as the safe synchronization point. This provides a speedup by preventing both simulators from having to perform a single step execution. If Chatoyant's next event time is a non-zero value, then both simulators will use this value for the next event time. This is seen in segment G for Chatoyant and segment H for ModelSim.

If the ModelSim next time is less than the Chatoyant next time, the simulators use this value for the next synchronization point. We do not need to perform the same check as in the previous case since ModelSim, according to the FLI reference manual (31) always returns a future event and if there are none, the default future event is set to the stop time of the simulation. Therefore, the simulators set the next synchronization point as seen in segments I in Chatoyant and J in ModelSim. If neither of the cases mentioned above are met, and since we assume that both simulators are at the same time step, we will manually step through the simulation using the SYNC_PULSE constant, seen in segment K. This segment is available only as a safety mechanism, since there should be always be an event on at least one queue until the stop time is reached.

The remainder of this section will describe the detailed implementation for both ModelSim and Chatoyant. The figures above are used as a simple representation of what each simulator must perform in terms of the co-simulation activities. The following sub-sections to provide us with a general idea of the structure of the code used in both Chatoyant and ModelSim. The code discussed in the following sub-sections is the template used by the generator to create all of the necessary files. After we discuss the general structure of the interface code, we then discuss the system generator which automatically creates all of these files.

4.2.1 Common Structures and Methods

There are a set of functions as well as data structures that are shared by both the Chatoyant star as well as the FLI code executed in ModelSim. Among the common functions are methods for opening, reading, writing, and closing the pair of named pipes that sit between the two simulators. These functions provide a simpler means of writing and reading data from the pipes. The shared structures include the interface port structure for event status update, bound for both simulators. There are a series of runtime constants used for decoding messages sent through the pipe as well as establishing the pipe handles. Finally a set of constants are available on a per system basis for the shared memory. These variables are the keys required to properly create and gain access to the allocated memory in both sides of the interface. The application of these structures and methods are explained in further detail in the following two sub-sections.

4.2.2 Chatoyant Interface Star

In this section we present the code that is generated for the customized interface stars in Chatoyant. The Chatoyant star needed for the co-simulation interface is created on a per-system basis. This is because of the high level of customization in terms of port names, voltage threshold values per port, and rise/fall time parameters per port needed. However, independent of the ports, is a consistent setup of the code in any Chatoyant interface star.

One of the first items that comprise a Ptolemy star is its title and domain. In this case, the name of the system is typically appended to the string *DEMTI_CHAT_PL_* to demark that this is a Discrete Event star for interface ModelSim (MTI) and Chatoyant (CHAT) in the Ptolemy description language (PL). The domain utilized in every interface star is discrete event or DE. Immediately following these definitions is a method call that allows this star to be a self-repeating star. This is a vital inclusion since it allows the star to reactivate itself at the next synchronization time however it is determined. Following this method call are extra comments, descriptions, and copyright notices which remain the same for all interface stars.

The next important element in the Chatoyant star is the header files that will be included in the generated CC and H files by the Chatoyant make program. For every system there is a custom header file generated usually of the form *MTI_CHAT_[System Name].h*. This file contains the shared memory structure definition as well as the shared memory key values and pipe file names. Also embedded in this header file is the callouts to all other header files needed for the general operation of both the Chatoyant star and the FLI code. This includes the OS level header files for instantiating the pipes and shared memory, as well as ModelSim and Ptolemy/Chatoyant specific functions.

The port instantiations follow all header file definitions. These ports define the Chatoyant-side signals that will cross over into ModelSim and back. These ports utilize the custom Chatoyant message type, and specifically, the electrical properties found within it. This message type creates either a single voltage or an array of voltages. Input ports define signals that are bound for ModelSim and likewise, ports designated as outputs become signals that originated in ModelSim and are bound for Chatoyant.

After the port declarations are the parameters for the star. A name, a type, and a short description define each parameter. The first parameter for the interface state defines the level of debug messaging. This value will range from 0, which defines quiescent, to 2, which defines verbose with 1 being a normal level of messaging. The second parameter defines the synchronization period. The scale for this parameter is on the order of picoseconds. This parameter is available for the lock-step synchronization method as well as a failsafe synchronization pulse period for the dynamic algorithm. The third parameter, the rise/fall time, determines the slope of the transition for an event change. This prevents Ptolemy from scheduling a zero time output event which could potentially crash the piece-wise linear circuit solver. The final parameter, the DE multiplier, determines the current time resolution of the simulation. The default value for this parameter is 1ps or 1E-12.

All globally accessible variables follow the parameter definitions. These variables are declared as private members when generated into C++ code. The variables declared here are grouped based on a common purpose. The first set of variables creates the previous values for all of the output ports, which permits creating a transition slope based on the rise/fall time parameter. These variables exists as electrical signals, as required by the Chatoyant Message type, as well as double precision floating points. There are also previous value variables for each input, which later determines if there is a true event change since it is possible to receive events of equal value at as the simulation time progresses. A discussion of this aspect will clarify this topic, discussed in this section.

The next set of variables hold the local instantiation of the pipe file descriptors. Following this is a pointer to the container galaxy's scheduler object. This is set for the dynamic scheduling routine. Next is a set of variables that hold Chatoyant's and ModelSim's current and next time values. These variables keep this time information from a past iteration. The last set of variables measure the number of events that pass through the interface as well as the number of synchronizations. The number of synchronizations is independent of the number of events since either algorithm synchronizes based on scheduling criteria and not event activity. Chapter 5 clarifies this point with real data from simulations.

The next section of this code is the constructor and destructor method definitions. These methods, adhering to the OOP mechanisms in C++, create and destroy all variables that are of the type Signal, found in the Chatoyant Message class.

With the variables and parameters defined, the remainder of the interface code for the star contains methods that perform some action during the simulation. The first of these methods are the voltage to logic conversion functions. There is a method to convert a voltage into a logic value for each input port. Likewise, there is a logic to voltage conversion function for every output port. Each port has its own threshold value set defined in the way seen in Chapter 3, Section 3.3.

One of the more vital functions is the *setup* method. This function is responsible for instantiating the scheduler pointer, if in dynamic mode, creating and opening the Chatoyant side of the two pipes, and initializing all other variables. First, the shared memory is created (if ModelSim has not already performed this operation) and the shared memory is setup for access via pointers to the common interface structures. Next, we create and open the pipes from the Chatoyant side. For the ModelSim bound port, the Chatoyant code is given write-only permission on the pipe. The interface gives the Chatoyant side of the interface read-only permission on the Chatoyant bound pipe. After this, we initialize all of the time keeping and event counting variables to 0. Finally, the values of the state saving variables are set to unknown and 0 volts.

Another function that is similar to the setup function, named *begin*, is called at every click of the Go button, found in the run window of Ptolemy. This function only re-initializes the variables of type Signal. This allows for back-to-back simulation capability. Opposite to the *begin* and *setup* functions is the *wrapup* function. This method behaves like the destructor method of the class, freeing and nullifying all of the Signal variables. A simulation ending at its stop-time calls the *wrapup* function.

The final and most critical method is the *go* function. This method is responsible for all simulation runtime execution. Whenever the scheduler sends an event to the porthole of the star, the go method executes to handle it. In the interface star, the *go* method is responsible for two actions: First, to update all port statuses and the second is to synchronize with ModelSim. The event communication scheme is consistent regardless of the synchronization scheme. As seen in Figure 17, the first step of the go method is the update of the input ports. This requires going through each port and determining if there is a change in state. Each bit of every port is checked to determine if there is such a change. If there is a change, then the port that has the change is set to dirty and the value in the shared memory is updated, after voltage to logic conversion. If there are no changes, a null is posted in the pipe, otherwise a ModelSim bound event is sent through the pipe. At this point Chatoyant waits for ModelSim to do the same on its end. When ModelSim posts either a Chatoyant bound event or a null, the code progresses to the synchronization section. It is at this point that code for the two synchronization approaches differ.

For the lock-step mode, the calculation of the next re-fire time, which will issue the scheduler a wakeup to call the star, is simply the current time (call *arrivalTime*) plus the synchronization pulse period. For the dynamic approach, the calculation requires extra steps. First, the next time must be fetched from the DE scheduler. This is available by a method added to the EventQueue data structure used in DE. The method added for the purpose of this interface searches the queue for the next highest timestamp, relative to the current time. When this time stamp is found, it is returned to the interface star and the value is posted to the shared memory. ModelSim performs the same operation, and based on the least value between these two times, the next re-fire time is set equal to the smallest time. At this point, all temporaries are made null, concluding the description of the Chatoyant co-simulation interface star. We now proceed to look at the ModelSim side of the co-simulation interface.

4.2.3 ModelSim FLI Interface

We now discuss the structure of the code, created by the generator, for the ModelSim side of the interface. The second half of the co-simulation interface is in special C code loaded in ModelSim. The code is built using ModelSim API functions from the Foreign Language Interface, or FLI. This FLI code is compiled and linked into a shared object file then called out in the architecture section of a VHDL design. The FLI provides numerous functions for getting information about a user’s design loaded within ModelSim. The FLI also provides state information such as current simulation time, delta cycle, and next event time, and is also capable of controlling ModelSim. These features make the FLI a suitable API for building the co-simulation interface since there is no possibility of linking directly into the ModelSim simulator core like Seamless.

The code created for the co-simulation interface shares some properties with the Chatoyant interface star. First of all, the FLI code is unique to every system due to its access to ports within Chatoyant. Likewise, there is also a similar pattern in terms of initialization, runtime, and stopping, as seen in the Chatoyant interface star. First are the global variables. First is a structure that contains the ModelSim FLI Signal for every port and the previous-state information for the Chatoyant bound ports. Following this structure definition is the declaration of the pipe file descriptors and the shared memory pointers, similar to the members in Chatoyant. The next item is the internal sync signal, used to schedule a wakeup in the ModelSim scheduler, similar to the re-fire in Chatoyant. This structure contains a signal ID (for getting its memory location), driver ID (for allowing the FLI code to force a new value), a process to attach itself to (similar to a VHDL process, except defined in the context of the FLI), and a last state variable. These will all be used when ModelSim determines when to call the synchronization routine. The next global definition is an enumeration of the IEEE Standard Logic values for fast lookup during runtime translation. The final set of variables is the event and null event counters the same as in Chatoyant.

Following the variable declarations is a set of utility functions. These functions provide a way to translate an enumerated value into a string of logic literals. This is necessary since ModelSim returns the current state of all signals as an enumerated value and not a string of characters. These functions utilize arrays defined the global declarations section.

Due to the rules of C and the inherent methodology of writing FLI code, the next section of code is the primary runtime function. This function, called *SyncInterface*, is the analog to the *go* method in the Chatoyant star. This method begins by retrieving the current ModelSim simulation time, and storing it in a temporary variable. Next, the function waits for the Chatoyant side to send any event update or null message information. These constitute the ModelSim bound ports. As soon as Chatoyant sends a message, ModelSim checks each dirty bit of the ModelSim bound ports. This updates any ports that have changed by issuing a force command similar to the way a force issued in ModelSim's GUI. ModelSim then determines if any of the ports bound for Chatoyant have changed state. If so, it marks the dirty bit for that port and updates the shared memory corresponding to that port. Once again, the code is relatively independent of the synchronization method used. Refer to Figure 17 for clarification.

With all event information communicated, ModelSim retrieves its next event time. If the synchronization method is dynamic, the FLI provides a predefined function for performing this operation. Therefore, unlike Chatoyant, where such a function had to be implemented in the Ptolemy DE scheduler, this method is part of the FLI. The next step is to get Chatoyant's *next event* time and then perform the same calculation that the Chatoyant interface star performs. Once the safe re-fire time has been determined, the code then makes use of the internal signal structure. This signal acts as a pseudo clock by toggling from '1' to '0' every re-fire time. If the past value was '1' then the wakeup schedules the signal to change from a '1' to a '0' at the desired time. If the synchronization method is lock-step, the signal is simply scheduled to wake the scheduler up at the sync pulse time. This is due to the fact that ModelSim works on the concept of relative time instead of absolute time.

With the conclusion of the main synchronization function, the next series of methods are similar to the *setup* and *wrapup* methods of Chatoyant. These functions operate as callback functions. The callbacks are not explicitly called at a known time. However, they are engaged when a certain state is reached within the ModelSim runtime environment. For example, if the simulator is closing, the quit callback linked in from the FLI code will be placed on ModelSim's runtime stack among the other closing procedures. The other callback that exists, other than the quit, is the restart. The quitting and restarting callbacks perform the same function as the quit by releasing all shared memory and pipes, however, this callback is only executed if the user restarts or closes a simulation in Modelsim.

The final function in the FLI code is the initialization function. This is utilized when ModelSim first elaborates a design and loads the shared object file. This method is specified in the VHDL container as the first method to execute. This method performs the same operation as the *setup* does in Chatoyant, but on the ModelSim side variables. The first step is creating and instantiating ModelSim's side of the shared memory. Following the shared memory initialization is the creation and opening of the pipes. This step is the reversal Chatoyant. ModelSim is the writer of the Chatoyant bound pipe and the reader of its incoming pipe.

Once the pipes are open, the generic values are captured. These values are set in the VHDL code calling the FLI shared object file, and are similar to the parameters found in the Chatoyant interface star. The only difference is that the debug message level and the synchronization pulse value are the only parameters available in the ModelSim side. The simulation resolution is handled externally in the ModelSim initialization file and there is no need for the rise/fall time parameter since discrete logic is used in ModelSim. The internal port structure members are then initialized. This follows suit as to what is found in the Chatoyant star.

Once the ports are initialized, the internal re-fire signal is instantiated and linked to the *SyncInterface* method. It should be noted that the *SyncInterface* is defined in the FLI code as a process and is viewed by ModelSim as a process equal to any VHDL process. Therefore, the FLI process maintains a sensitivity list of signals that can activate its execution. The default for the *SyncInterface* process is the internal re-fire signal. Directly after the re-fire signal is initialized, the Chatoyant bound signals are added to the *SyncInterface* process's sensitivity list. These aide ModelSim in maintaining consistency with Chatoyant with regards to the criteria based in the synchronization routines. These criteria are the simulators respective re-fire signal or event and the ports bound to the other simulator. During simulations, the preceding criteria allow for 100% accuracy by providing dynamic synchronization.

The final step in the initialization function is the instantiation of the callback methods. This is accomplished by a methodology that allows FLI code to add the desired methods to the proper callback list. It is at this time that all event and message counters are set to zero. With that last step, the FLI code is completed. With an understanding of the code structure for the interface, we can now discuss the system generator which creates all of the code needed for operation. This automatic generator is explained in more detail in the following section.

4.3 AUTOMATED CO-SIMULATION INTERFACE GENERATOR

As previously discussed, despite the apparent reusability of much of the code in both the Chatoyant star and the ModelSim FLI code, it is still necessary that customization be done within the co-simulation environment. The driving factor behind this inherent uniqueness is the fact that all systems differ in the ports that cross the interface boundary. Much of the code in both sides of the interface could be condensed down into a general template. However, the event handling sections demand the most attention for an automation system.

To reduce error and system setup time, the following flow was implemented in Figure 19. This flow utilizes a co-simulation interface generator that allows the user the ability to select ports that are included in the interface. From that selection it will generate all of the necessary files and libraries needed by both Chatoyant and ModelSim. Figure 19 shows this flow and what feeds the generator and what outputs are needed by each simulator.

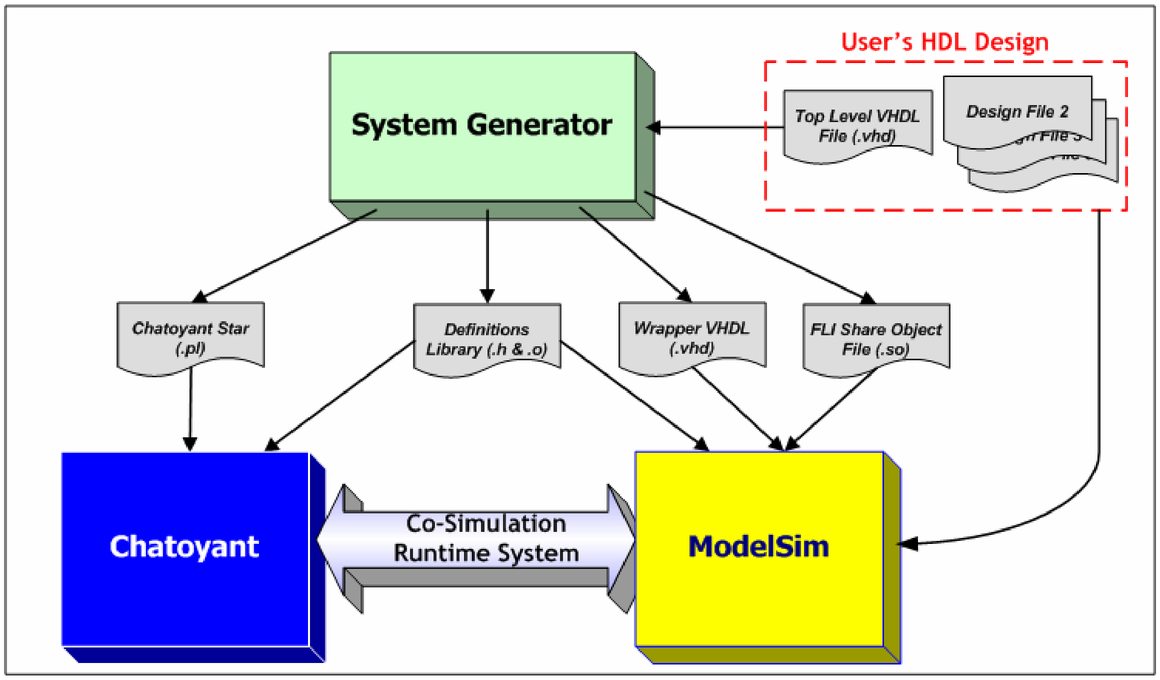
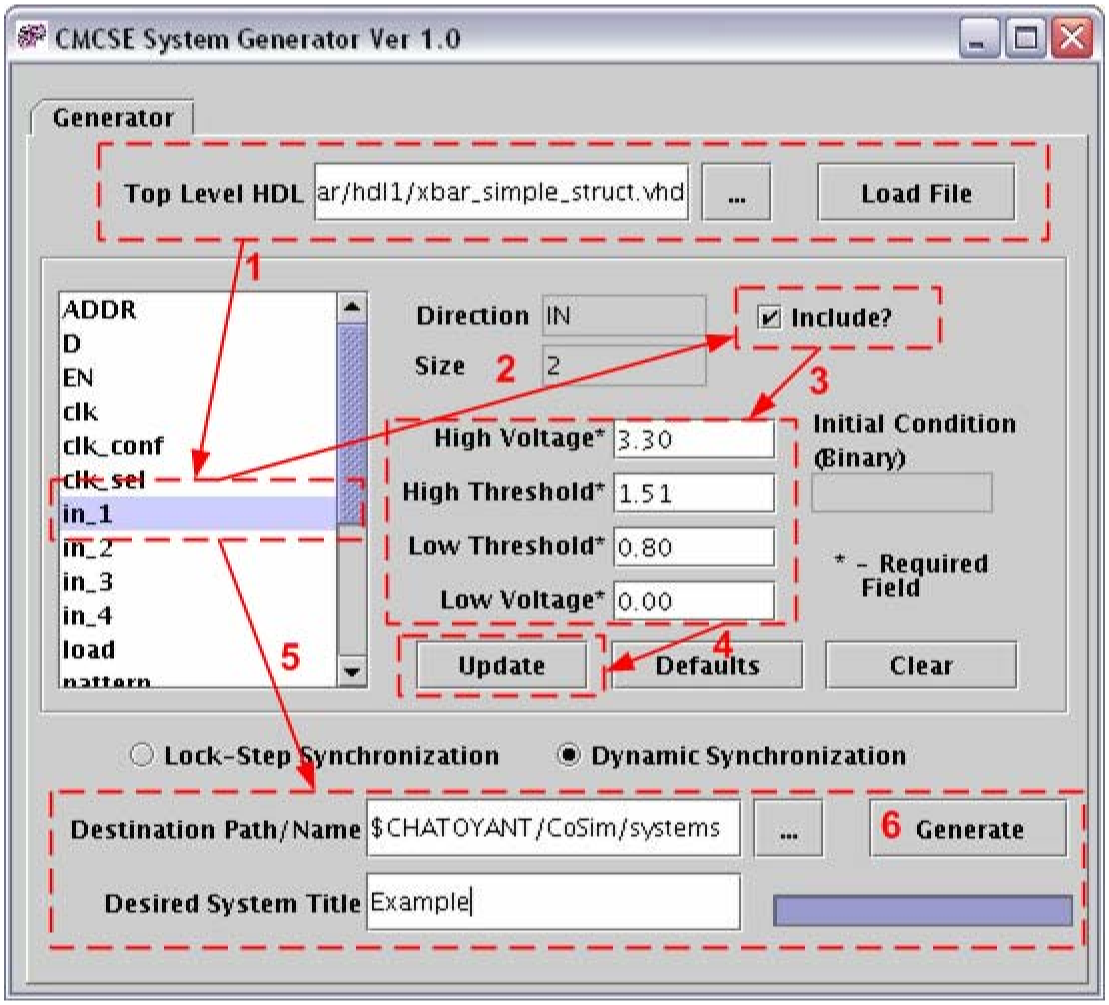

User's HDL Design
System Generator
Top Level VHDL File (.vhd)
Design File 2
Chatoyant Star (.pl)
Definitions Library (.h & .o)
Wrapper VHDL (.vhd)
FLI Share Object File (.so)
Chatoyant
Co-Simulation Runtime System
ModelSim

Figure 19 Co-Simulation system flow with the generator

As seen in Figure 19, the key to generating the correct files for the co-simulation interface is the user's HDL design, namely the top-level of the design or part of the design that will be interfacing with Chatoyant. Using the generator GUI, seen in Figure 20, the user can select the VHDL file that contains the ports they want included in the interface. The generator proceeds to parse the entity section of the selected VHDL file. During the parsing, a list of ports are created, which includes each port' name, direction, vector size (if not a single bit), and initial value.

Once this list is complete, it is displayed in a list-field in the GUI and the user is able to select which ports to include in the interface (transition number 1 in Figure 20). A decision to include or exclude is made at this time (transition number 2 in Figure 20) and each port included must be assigned a voltage for each of the four thresholds (transition number 3 in Figure 20). Once selections have been made, then they are updated (transition number 4 in Figure 20). A destination directory and a name for the system must be specified to generate the files (transition number 5 in Figure 20). The final operation of pressing the generate button (item number 6 in Figure 20) will produce all of the necessary files needed by both Chatoyant and ModelSim.


CMCSE System Generator Ver 1.0

Top Level HDL: ar/hdl1/xbar_simple_struct.vhd

ADDR
D
EN
clk
clk_conf
clk_sel
in_1
in_2
in_3
in_4
load
pattern

Direction: IN
Size: 2
Include? [✓]
High Voltage*: 3.30
High Threshold*: 1.51
Low Threshold*: 0.80
Low Voltage*: 0.00
Initial Condition (Binary):
* - Required Field

○ Lock-Step Synchronization [● Dynamic Synchronization]

Destination Path/Name: $CHATOYANT/CoSim/systems
Desired System Title: Example

Figure 20 Chatoyant-ModelSim Co-Simulation Environment System Generator GUI snapshot

There are a set of files produced by the generation process. Included is the .pl file for Chatoyant, the .c file for the FLI, the common .h and .c file for shared structures, constants, and functions, a Makefile for compiling the FLI and shared library files, the FLI wrapper VHDL file, a top-level VHDL file that links the user's top-level design and the FLI wrapper together, and a Readme.txt for explaining all of the files and providing a summary of the generator's actions.

When all of the files have been compiled from the Makefile, the user must add the .pl file to Chatoyant using the proper method of linking a star into Chatoyant. The VHDL files and the

.so file created must be appended to the user’s design and compiled into ModelSim. With those tasks performed, the user is now able to co-simulate their design. Subsequent chapters discuss how applications that utilize this flow to create series of designs that will measure certain aspects of performance within the co-simulation interface.

This chapter explained how the co-simulation environment between ModelSim and Chatoyant was created. This chapter dealt with the makings of the co-simulation interface as a new feature for Chatoyant. This included the operating system level constructs used for communication as well as the higher level algorithms that implement the time and event synchronizations. The following chapter describes six unique systems, all of which test a particular aspect of the co-simulation environment. Performance measurements are made and finally a set of conclusions are drawn.

5.0 SYSTEM APPLICATIONS

5.1 INTRODUCTION

With the theory and implementation of the co-simulation interface explained in the previous two chapters, this chapter shows several applications of the interface. First, a set of test systems are examined, which stress different aspects of the interface, in both modes of synchronization. In addition to these simple tests are two complex systems currently being modeled and implemented. These systems contain some elements of VHDL code, analog circuits, and optics (free space or fiber based). For the scope of this research, we limit the performance analysis to that of the co-simulation interface and minimal support components in both ends of the interface. This requires using only source and graph stars within Chatoyant and the minimal design in VHDL.

The first system represents a simple one-way link from ModelSim to Chatoyant. This is in the form of behavioral VHDL producing a clock pulse that is 200MHz from ModelSim to Chatoyant, which has a display graph to verify that the output is at the proper frequency and time for transitions. This system is tested using both the lock-step and dynamic synchronization schemes. The lock-step method is tested at 1ps and 1ns to show any speed-up between these two resolutions. We will also see if this cycle-accurate mode has any impact on the number of events seen across the interface.

The second system is similar to the first, however, executes in the reverse with the source being Chatoyant and the destination being in ModelSim. Once again, event data and waveforms are used to determine correctness and accuracy. These two systems represent simple operation with single, one-way event flow. This is significant to show basic operation and accuracy. It will also show the impact on the interface when a clock or periodic signal crosses over the interface.

The next set of test systems exercise the interface's multi-bit operation. Similar to the first two examples, there is one system allowing ModelSim to be the source and Chatoyant the data sink. Likewise, the other system is setup to have Chatoyant be the source of data and ModelSim the sink. Once again, accuracy concerning the ordering of the bits in signal arrays and timestamp of events are verified. With these two systems, a comparison is made with regards to the performance of the lock-step versus the dynamic synchronization algorithm, when considering the behavior of the events crossing the interface.

The next system evaluated represents the infrastructure of a currently modeled design. This design, known as Fiber Image Guide, or FIG, utilizes fiber optics, analog drivers and receivers, and digital control hardware. For the scope of this research, the co-simulation interface contains the digital circuits, described in structural RTL (Register Transfer Level) VHDL, and the other side of the interface is concerned with only supplying and graphing within Chatoyant. The uniqueness of FIG lies in that it exercises both directions of event flow, namely Chatoyant being both the data source as well the data sink, with ModelSim being a middle component.

The final system observed is the Smart Pixel Optical Transceiver, or SPOT. This system provides a variation to FIG in that it contains a series of single signals, one being a clock and the others being data, all crossing the interface It also requires ModelSim to be the source and sink of all event data, with Chatoyant being the intermediary.

For each system, we present a brief description of its design and relevance to defining performance and validation measurement of the co-simulation environment. The performance metrics discussed in Chapter 3 are applied to these systems, and items such as waveform and numerical output as examined for simulation accuracy. For all of the systems, the following runtime parameters are used. First, the workstation that these simulations were performed on was a Dell Precision with dual 2.2 GHz Intel Xeon processors with 2GB of RAM. The Operating System used was Red Hat Linux 7.3 using the core 2.4 Linux kernel and GCC 3.2.3 for compilation. For all systems, the system co-simulation interface generator, discussed in the previous chapter, was used to create both the lock-step and dynamically synchronized interfaces. The latest version of ModelSim, 5.8c was used for all VHDL simulation. Wherever possible, behavioral coding constructs were used in order to exploit the maximum performance from ModelSim. Ptolemy Classic, version 0.7, was used for the underlying framework and Chatoyant version 1.2 was used as the main simulation system. Shown in all results are the average of three executions of every system, and for the lock-step systems, three runs each at 1ps and 1ns were performed. Graphical plots are provided as well as runtime measurements and system profiling when available.

5.2 SINGLE BIT, MODELSIM TO CHATOYANT APPLICATION

5.2.1 System Description

The first system that is evaluated is the single bit, clocked signal originating in ModelSim and terminating in Chatoyant. This system, as done in the remainder of the systems discussed, is split into three segments. The first is the ModelSim side of the interface which contains the VHDL code. The second section includes the Chatoyant stars that are separate from the interface star. The third segment is the interface itself, which contains the FLI code for ModelSim and the Chatoyant star, as well as the logical connection between the two simulators.

As for the ModelSim side, behavioral VHDL is utilized to create a self oscillating clock. Figure 21 shows the graphical description of this code using the latest HDL Designer Series graphical code entry application from Mentor Graphics (version 2004.10). Shown in this diagram is the initial block and then a forever loop that will wait for one-half a clock cycle, then set the clock to a '1' value, wait for other half of the clock cycle, and then negate the signal back to '0'. The clock cycle has a period of 5ns or 200MHz.

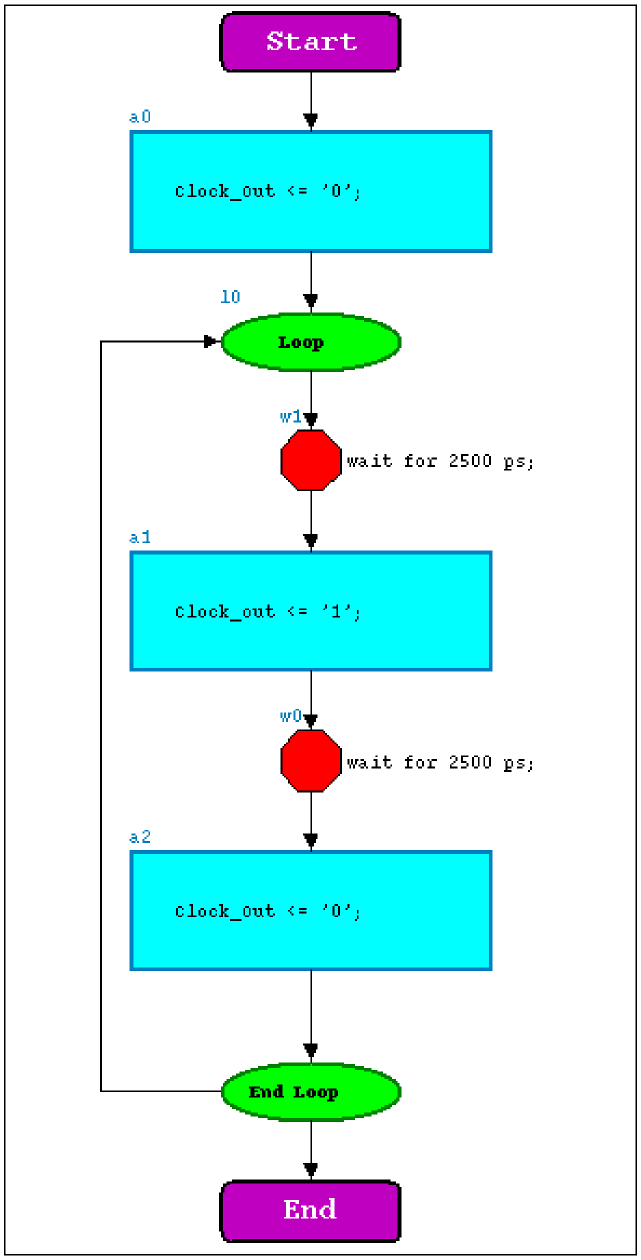


Figure 21 ModelSim behavioral VHDL for the single bit clock from ModelSim to Chatoyant

The Chatoyant side is relatively simple in that it contains the interface star as well as a bit-to-graph splitter and a plotting star. All graphical data is stored in a file for later examination and to also reduce run-time drawing overhead. Figure 22 shows the Chatoyant galaxy used for the Single bit test. Though the galaxy for the lock-step and dynamic synchronization are the same structure, they are two unique palettes due to the static compile–time nature of selecting which algorithm to use.

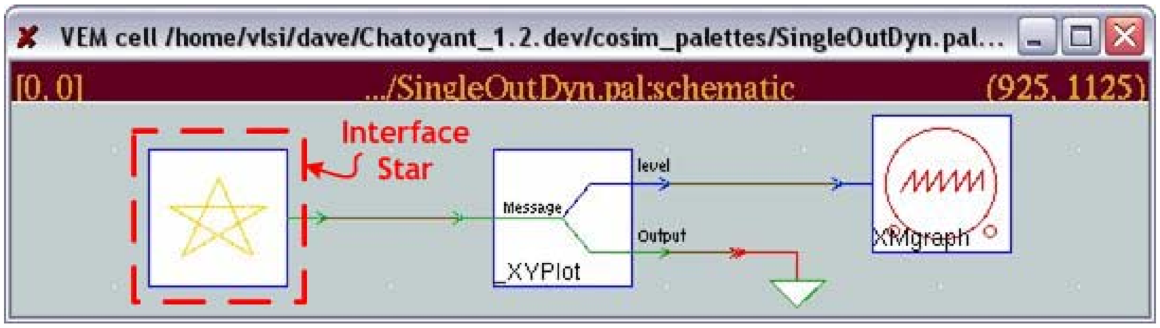

VEM cell /home/vlsi/dave/Chatoyant_1.2.dev/cosim_palettes/SingleOutDyn.pal...
[0, 0]
.../SingleOutDyn.pal:schematic
(925, 1125)
Interface Star
Message
level
Output
XYPlot
XMgraph

Figure 22 The Chatoyant side of the single-bit from ModelSim (Dynamic system shown)

The co-simulation interface, for this system, remains simple, due the single bit and single port structure. For the FLI code, the two algorithms are implemented as two separate interfaces, with two separate wrappers. The code for each follows the structure set in the previous chapter. For the event communication, there is only one port sensitized to the synchronization process in addition to the internal FLI signal. Within Chatoyant, the block seen on the left hand side of Figure 22 is generated twice, each instance for both of the algorithms tested, with each instance placed in it own unique galaxy. This star has only one output to consider and the remainder of the code is synchronization.

This system, though small, stresses the system in terms of seeing how fast ModelSim can feed events to Chatoyant, and how fast Chatoyant can consume these events. This system also tests to ensure that events occur when they are supposed to on each side of the interface as well as verifying that the voltage conversion process is performed correctly.

5.2.2 Runtime Results and Analysis

There are several measurements to examine when comparing all synchronization methods. Appendix B provides the complete runtime results collected for this system. First, we must compare the co-simulation performance to that of a simulation that contains no co-simulation. In this particular case, this is limited to ModelSim's runtime since it is the only source of events. The runtime for this system, executing without any co-simulation interface, is 32 milliseconds. This runtime is equal to 12,500 events per second, assuming 400 events, 1 event per rising edge and 1 event per falling edge.

As seen in Figures 23, the events per second, ranked from fastest to slowest, is first the non co-simulation, secondly, the dynamic co-simulation, thirdly, the 1ns synchronization resolution for the lock-step mode, and finally, the 1ps resolution lock-step mode. This measurement is for the events that traverse the interface, in the cases of the co-simulation measurements, and not the total events in a particular simulator

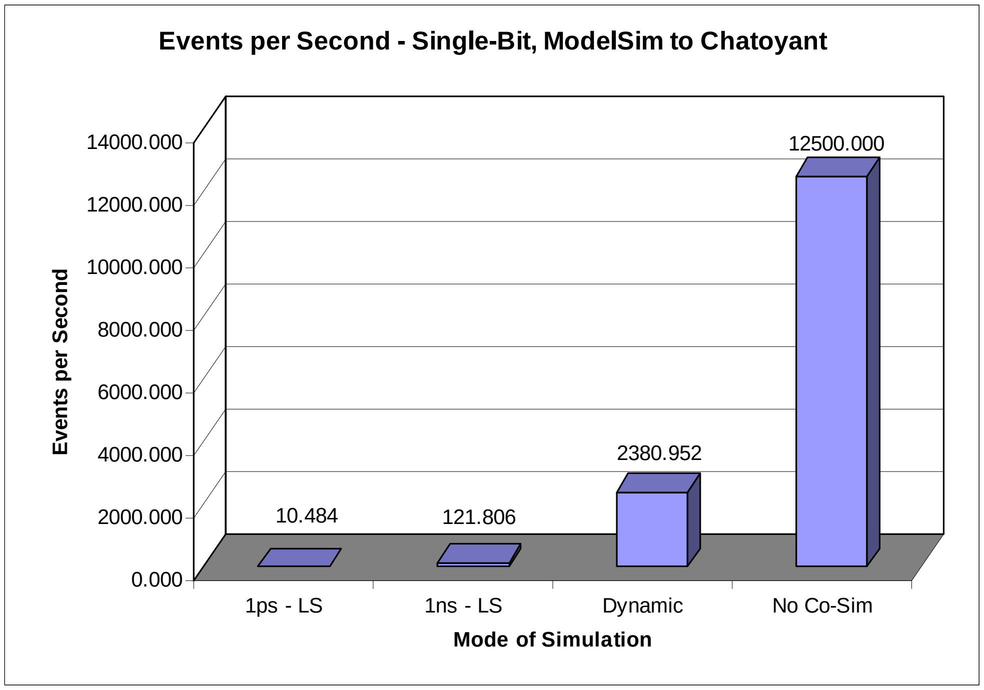


Figure 23 The events per second for the single-bit, ModelSim to Chatoyant test system

The non co-simulation is 5.25 times faster for the non co-simulated system versus the dynamic synchronization. The dynamic mode is 14 times faster than the 1ns synchronization resolution and 227 times faster than the 1ps resolution runtime. In terms of events per second, the non-co-simulation execution is once again 5.25 times faster than the dynamic mode. Likewise, the dynamic mode is 19.5 times faster than the 1ns lock-step resolution and 227 times faster than the 1ps lock-step resolution.

The primary focus for this thesis is the examination of the dynamic synchronization results. We focus on this specific mode since it provides the best overall performance for co-simulation. When we analyze this mode of synchronization, we can observe several characteristics. First, we see the presence of null events, or events that indicate just synchronization has occurred without any real event communication. These null events are present in dynamic mode since for every event sent through the interface from ModelSim, there are two Chatoyant events generated. These events will cause the lookahead function within the DE scheduler to find two additional synchronization points when the interface code enters its synchronization section. In addition to this, the interface star will also schedule one other synchronization if the event queue is empty, which it is when the real event has finished changing the state of the ports exiting the interface star (i.e., a rising edge or falling edge for a port value). The overall number of null events present is equal to the 1ns lock-step mode. The number of real events that cross the interface is equal to the 1ps lock-step mode, since the 1ps lock-step resolution allows all of the transitions to be seen, unlike the 1ns resolution, as seen in Appendix B.

It is seen that most of the overhead in Chatoyant is not in the synchronization or interface code. As shown in the pie graph in Figure 24 most of the time in Chatoyant is spent processing events. As seen in three of the other systems discussed later, this is most likely due to the null events being processed instead of Chatoyant having to processes only the real events.

This pie chart, as seen in all of the systems discussed in this chapter regarding the Chatoyant processing distribution, is divided into 7 categories. The first 5 of these categories relate to the co-simulation interface methods. This includes the main go() method where most of the interface operation is found, the read_fifo() and the write_fifo() methods which perform the pipe communication during synchronization, and the conv_to_logic() and conv_to_volt() which are the functions that convert between voltages and logic values. The next measurement is the event processing slice. This is the sum of all functions called during simulation that are part of the event handling process. This includes the scheduler actions and accesses to the event queue, handling events at portholes (both sending and receiving), and other simulation control associated with a simulation. Appendix B shows a breakdown of all event processing-based functions. The remainder of the graph shows the other processing performed by Chatoyant. This includes the setup of the universe, galaxy, stars, as well as the execution of the other stars independent of the interface star.

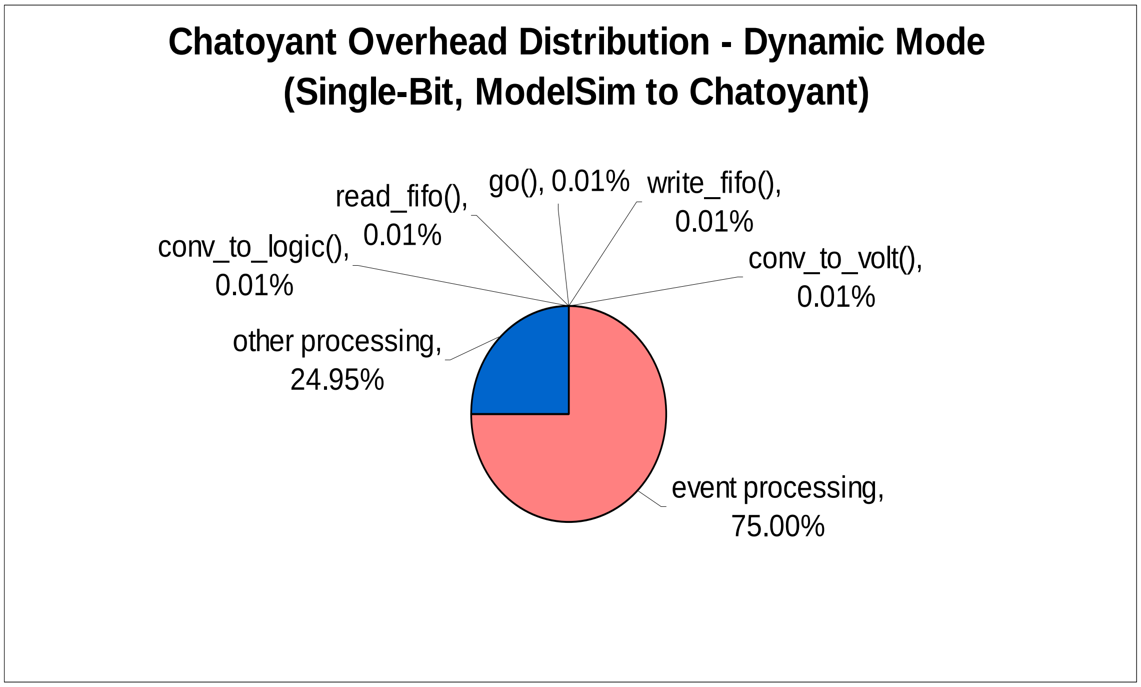


Figure 24 The Chatoyant runtime distribution graph for the dynamic synchronization co-simulation

The high Chatoyant event processing overhead is most likely due to the DE scheduler of Ptolemy utilizing a linear queue instead of a priority queue, where the algorithmic performance is O(*n*) instead of the sub-linear O(log *n*) as in a priority queue, where *n* is the number of events processed during every iteration (9).

Within ModelSim, the synchronization overhead accounts for half of the execution time. This is a higher distribution than seen in Chatoyant, due to the lower event processing overhead seen in ModelSim. The ModelSim execution distribution is seen in the pie chart of Figure 25.

This graph, like the others that are seen throughout this chapter, shows six possible slices. The first five slices constitute the co-simulation interface function calls. The most important of these five functions is the SyncInterface() method, which is where most of the synchronization is performed. The next two functions, write_fifo() and read_fifo(), are the function calls made by the SyncInterface() function, to access the pipes between ModelSim and Chatoyant. The last two function calls, conv_to_string() and conv_to_enum(), are the functions that convert between the string form and the enumerated form of the signals within ModelSim. The final section shows on the processing chart accounts for all other processing performed by ModelSim.

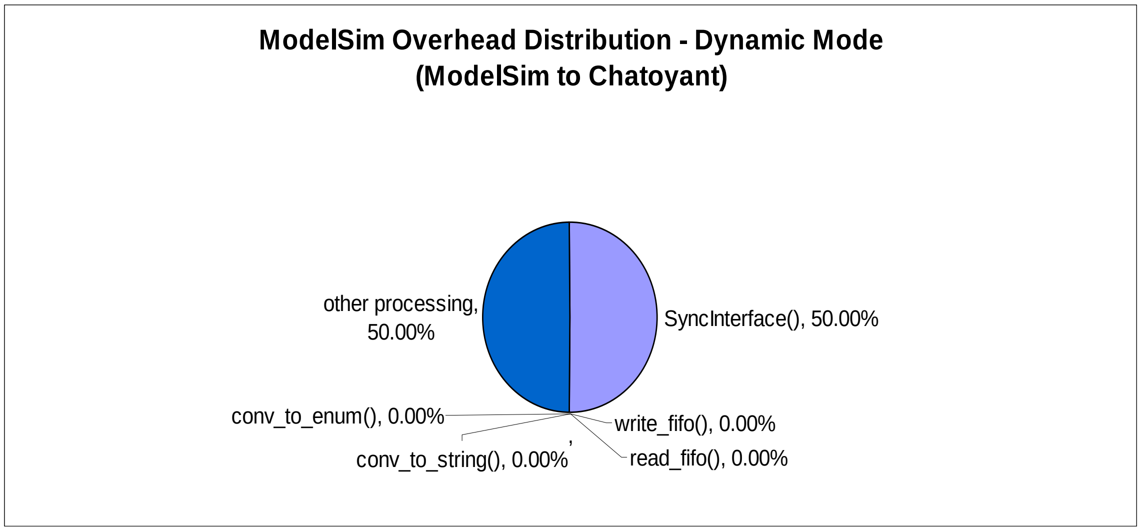


Figure 25 The ModelSim overhead distribution graph for the dynamic co-simulation

Finally we examine the waveforms for this system in the case of dynamic co-simulation. Figure 26 shows the ModelSim waveform as it enters the co-simulation interface. In this waveform we see s transition at 2.5ns.

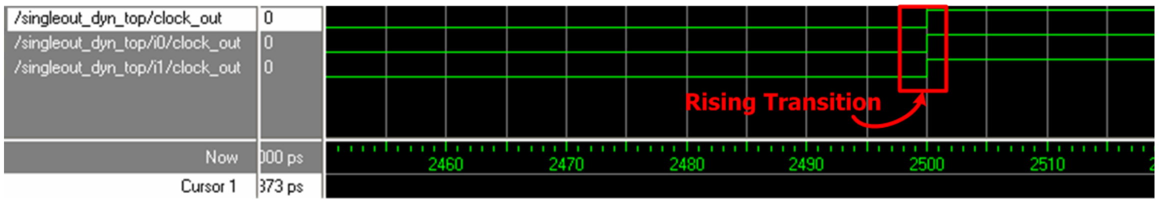

/singleout_dyn_top/clock_out
/singleout_dyn_top/i0/clock_out
/singleout_dyn_top/i1/clock_out
0
0
0
Rising Transition
Now
000 ps
Cursor 1
373 ps
2460
2470
2480
2490
2500
2510

Figure 26 ModelSim waveform for the single-bit, ModelSim to Chatoyant system in dynamic mode

In Figure 27 we see the resulting waveform in Chatoyant. In this figure we zoom in on the 2.5ns region to show the rise time slope that is added in the interface. This slope is present so that stars in the electrical domain, such as the piece-wise linear solver, do not receive an instantaneous transition. Such a transition is not handled within the solver. Therefore, the transitions are sloped by a factor, set by the user in the interface's global parameters.

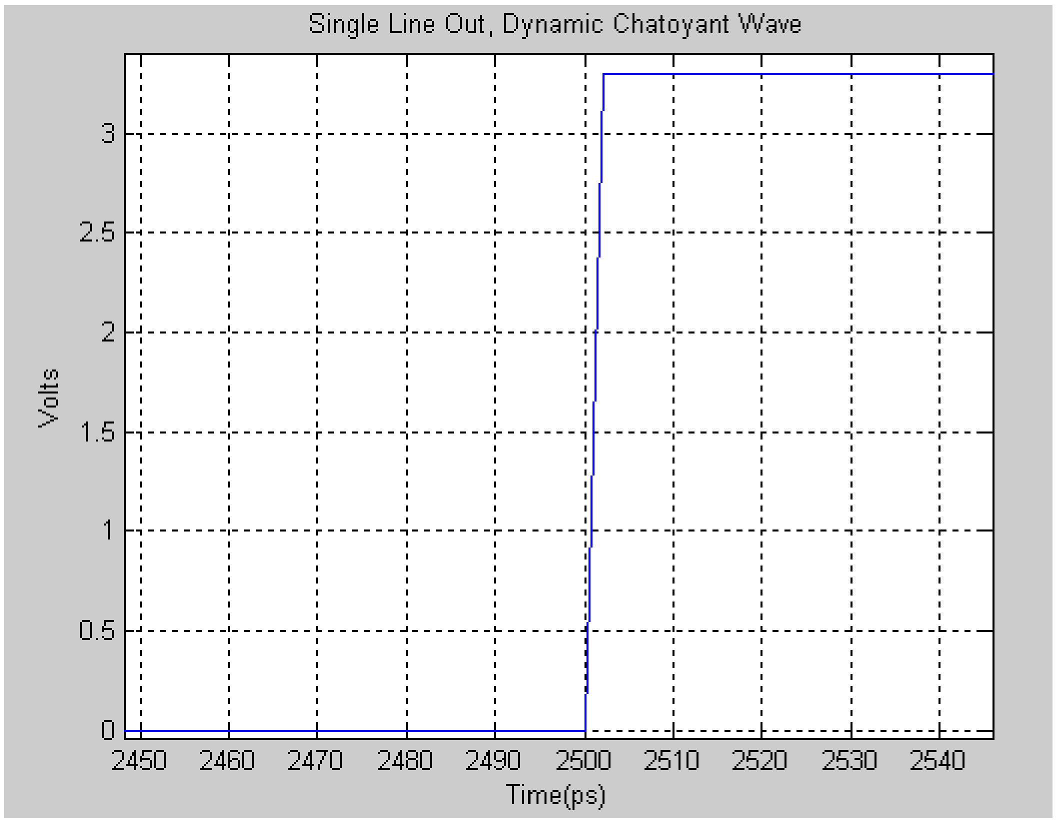


Figure 27 Chatoyant waveform for the single-bit, ModelSim to Chatoyant co-simulation in dynamic mode

At the end of this chapter we discuss, the implications of the results found within this section compared to the other systems discussed in the following sections. This cross comparison provides a look into how the co-simulation scales and performs for larger systems.

5.3 SINGLE BIT, CHATOYANT TO MODELSIM APPLICATION

5.3.1 System Description

The second system we examine is similar to previous one, with the exception that the events now originate in Chatoyant and terminate in ModelSim. Utilizing the piece-wise linear source star found in the Chatoyant electrical library, the same 200MHz clock signal can be generated as seen in the previous system. The piece-wise linear source star is a simple script reader, reading an ASCII text file that contains a list of times and voltage values. In ModelSim there is behavioral VHDL code that takes the single-bit signal that crosses the interface and then uses it as a clock to an eight bit counter. The result is two 4-bit buses, one bus is the high nibble and the other bus is the low nibble. Figure 28 illustrates the Chatoyant palette for this test system and Figure 29 shows the VHDL source for the code loaded into ModelSim.

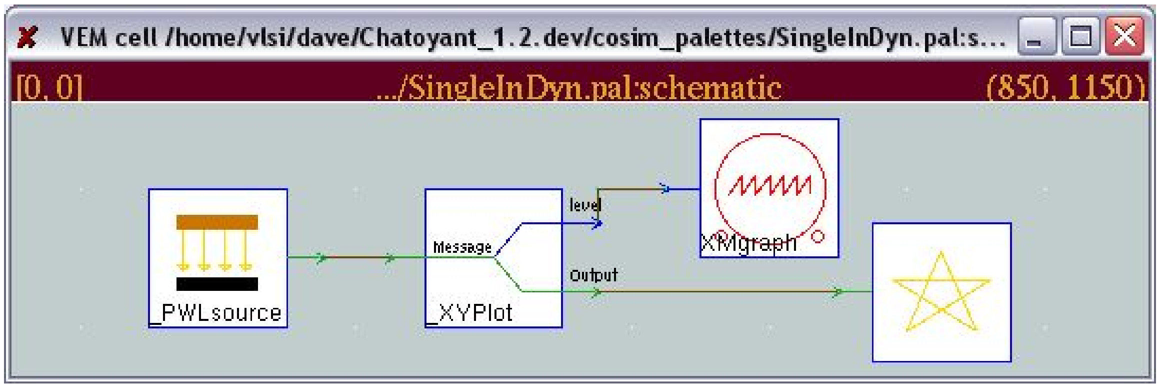
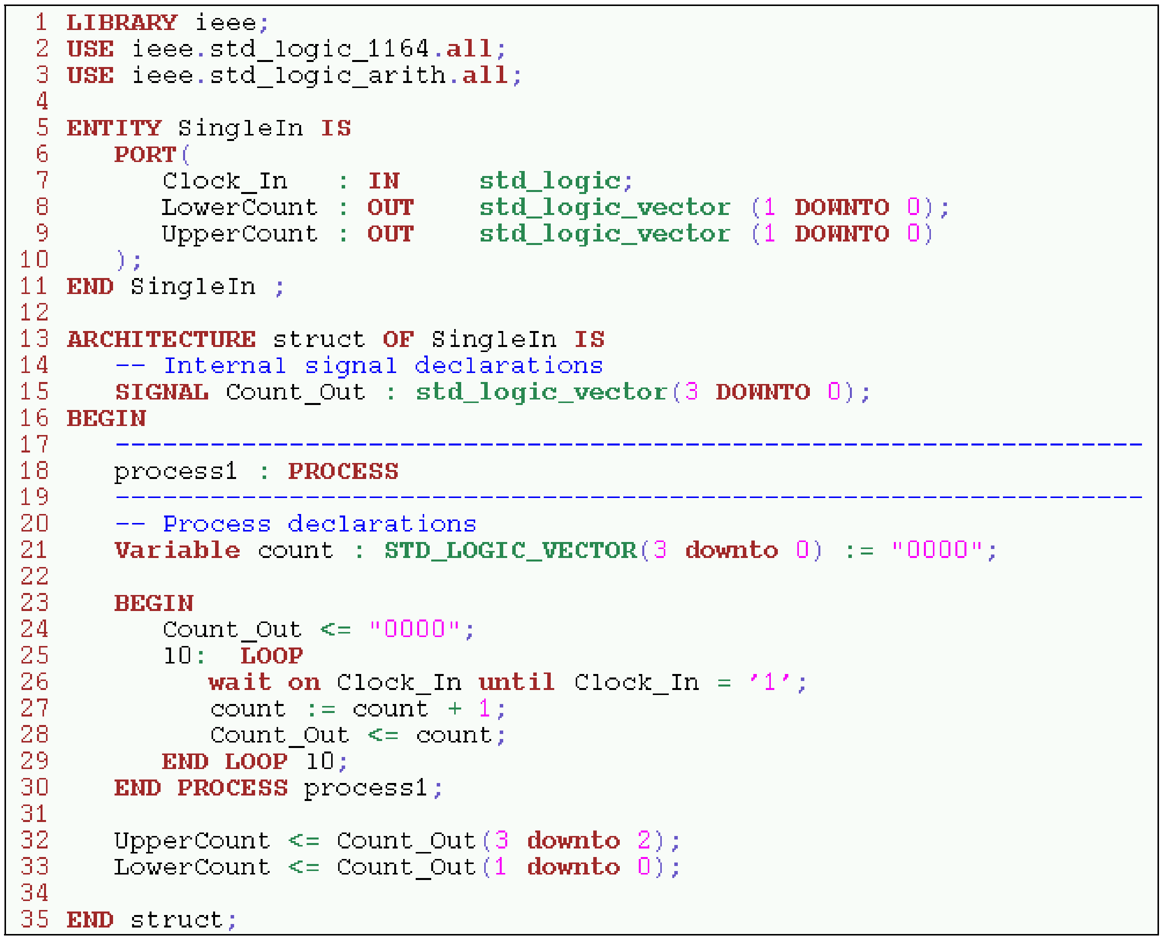

VEM cell /home/vlsi/dave/Chatoyant_1.2.dev/cosim_palettes/SingleInDyn.pal:s...
[0, 0]
.../SingleInDyn.pal:schematic
(850, 1150)
_PWLsource
Message
_XYPlot
level
Output
XMgraph

Figure 28 The Chatoyant palette for the single-bit, Chatoyant to ModelSim test system



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY SingleIn IS
   PORT(
      Clock_In    : IN     std_logic;
      LowerCount  : OUT    std_logic_vector (1 DOWNTO 0);
      UpperCount  : OUT    std_logic_vector (1 DOWNTO 0)
   );
END SingleIn ;

ARCHITECTURE struct OF SingleIn IS
   -- Internal signal declarations
   SIGNAL Count_Out : std_logic_vector(3 DOWNTO 0);
BEGIN
   -----------------------------------------------------------------
   process1 : PROCESS
   -----------------------------------------------------------------
   -- Process declarations
   Variable count : STD_LOGIC_VECTOR(3 downto 0) := "0000";

   BEGIN
      Count_Out <= "0000";
      l0:  LOOP
         wait on Clock_In until Clock_In = '1';
         count := count + 1;
         Count_Out <= count;
      END LOOP l0;
   END PROCESS process1;

   UpperCount <= Count_Out(3 downto 2);
   LowerCount <= Count_Out(1 downto 0);

END struct;
```

Figure 29 VHDL source code for the counter that will use the single bit from the interface

As the single-bit, ModelSim to Chatoyant example showed how the interface performs when ModelSim is the source of events, this system tests to determine how the interface performs when Chatoyant is the source of events. We also see how ModelSim performs converting these incoming events into signal "forces". As in the previous system, the central clock operates at 200MHz, which for a 1us simulation time is 200 cycles. We present the data from the lock-step and dynamic synchronization modes in the following sub-section.

5.3.2 Runtime Results and Analysis

As in the previous section, we provide more thorough listing of all runtime data in Appendix C for this system. We first execute the non co-simulation system to gain a baseline for the performance of this particular system. In this case, the events originate in Chatoyant, therefore we must run the system with all of the stars present in the co-simulation version, without the interface star. Chatoyant performed the simulation without the interface in 385 milliseconds. This is equal to 1039 events per second, assuming 400 events (200 rising edge events and 200 falling edge events for 200 total cycles).

Figure 30 shows a comparison graph between the lock-step synchronization execution, the dynamic synchronization execution, and the non co-simulation execution in terms of the events per second.

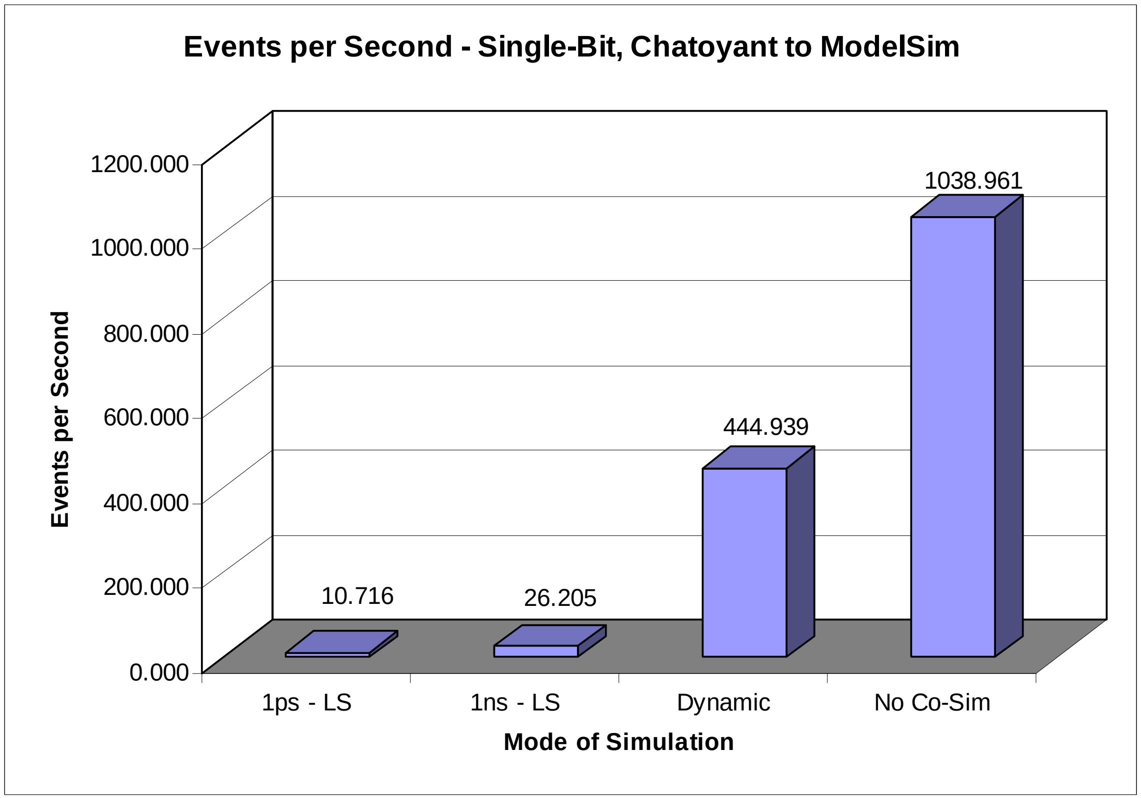


Figure 30 The events per second for the single-bit, Chatoyant to ModelSim execution modes

In terms of events per second execution performance, the dynamic synchronization provides between an 8.55 times (1ps) to a 10.43 time (1ns) speedup over the lock-step modes. Overall, the non co-simulation execution is 4.3 times faster than the dynamic co-simulation.

Further analysis into the dynamic synchronization mode is found in the profile of the code for both the Chatoyant and ModelSim side of the interface. Figures 31 and 32 are the Chatoyant and ModelSim respective pie charts showing the breakdown of execution.

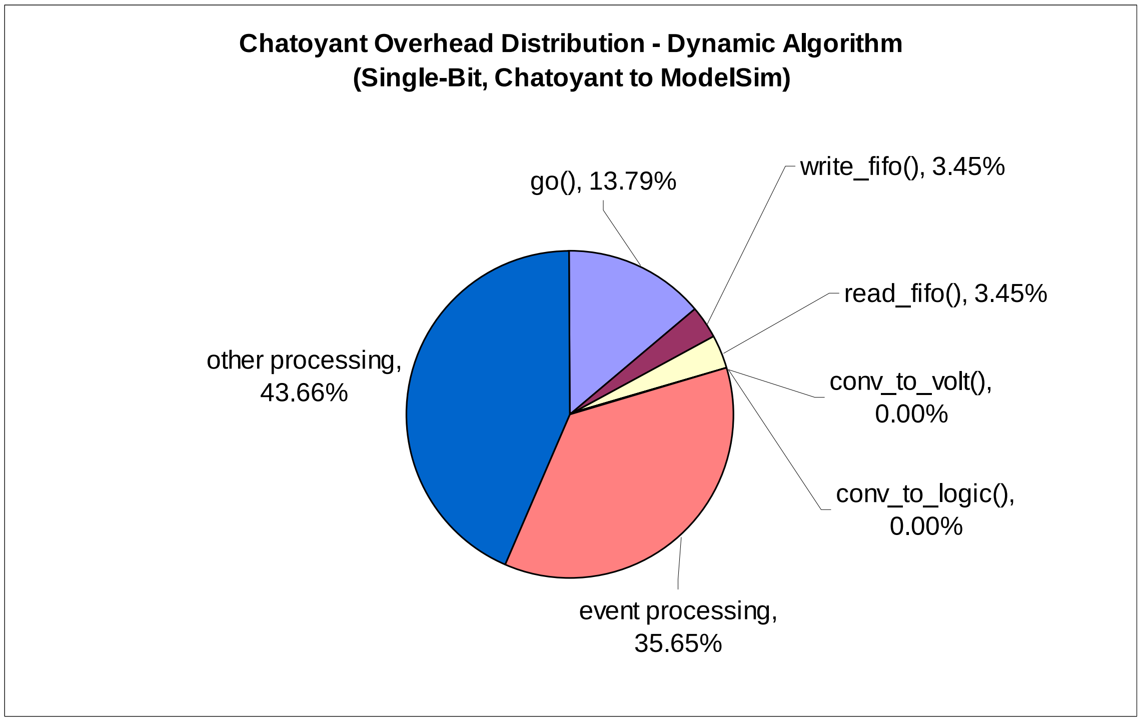


Figure 31 The Chatoyant code profile using the dynamic synchronization

The graph seen in Figure 31 shows that most of the execution time is found in the other processing within Chatoyant, namely the piece-wise linear source and graphing stars. The second most amount of time is spent within the event processing functions within Chatoyant. Thirdly is the execution time of the *go()* method within the Chatoyant interface star. This shows that the interface is a significant contributor to the execution time of the simulation, but the amount of event processing is still higher than the interface. The piecewise linear source star's go() method accounted for 1/3rd of the overall processing time, in addition to approximately the other 10% of functions neither related to the co-simulation interface or event processing.

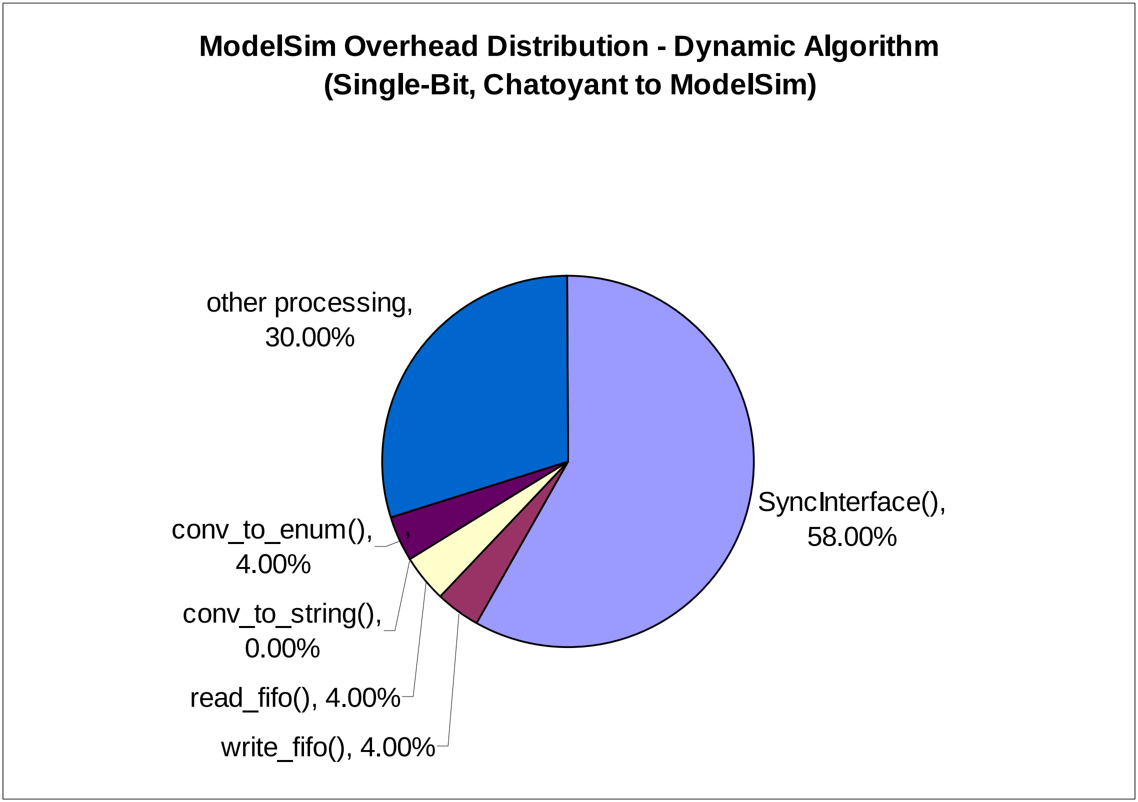


Figure 32 The ModelSim code profile for the dynamic synchronization

Within the ModelSim code profile, the *SyncInterface()* method consumes a majority of the execution time. The other processing performed by ModelSim consumes only 20% of the total execution time, implying that the interface code is the largest bottleneck in the simulation on the ModelSim side of the interface.

This is seen in Figures 33, for the Chatoyant side of the signal, and 34, for the ModelSim waveform of the received signal.

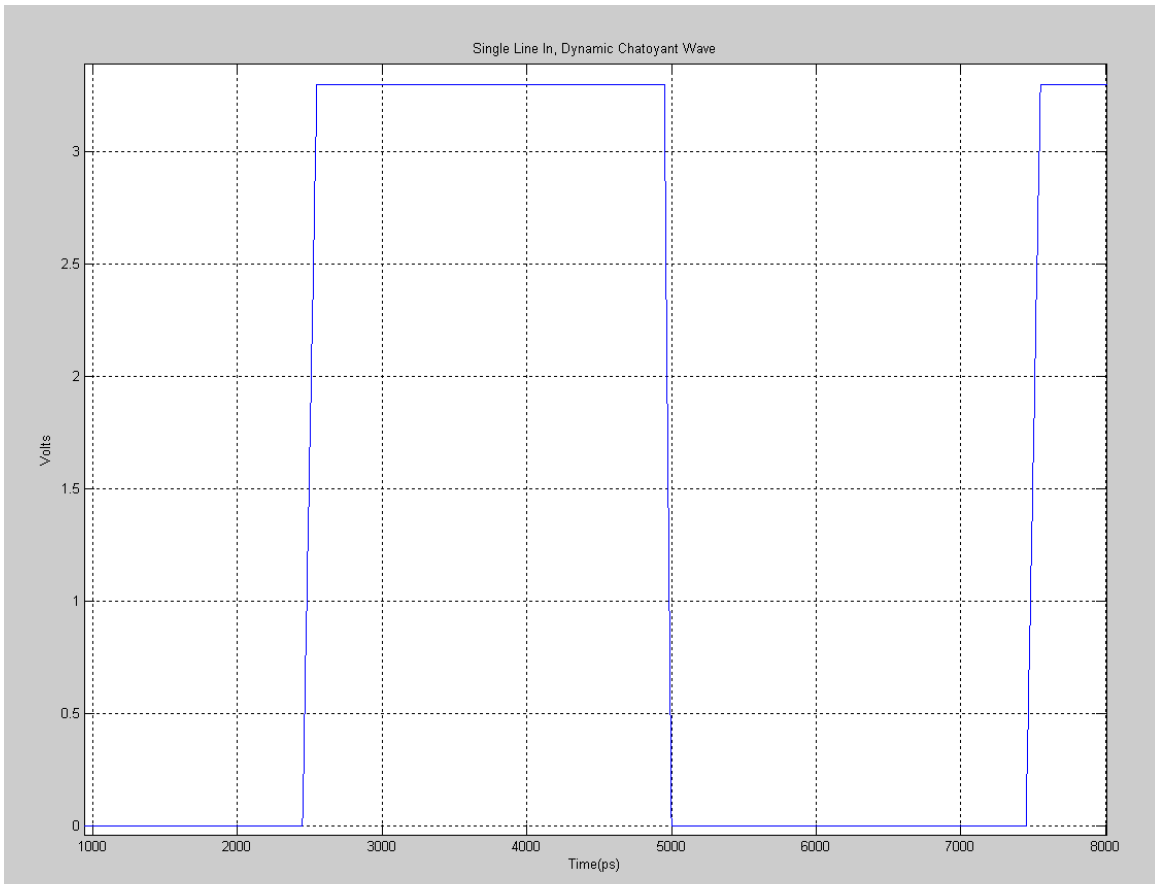
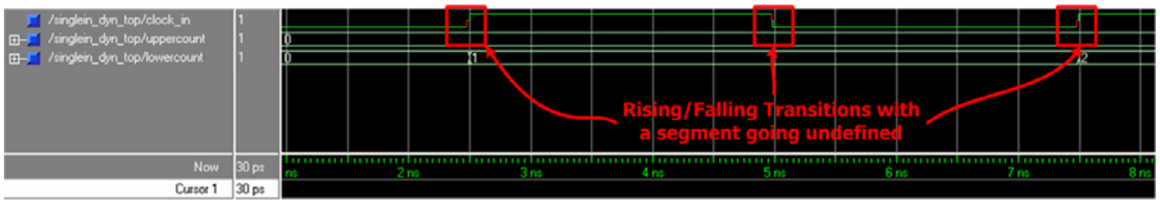


Figure 33 The Chatoyant waveform using the dynamic synchronization mode



/singlein_dyn_top/clock_in
/singlein_dyn_top/uppercount
/singlein_dyn_top/lowercount
Rising/Falling Transitions with a segment going undefined
Now 30 ps
Cursor 1 30 ps
2 ns
3 ns
4 ns
5 ns
6 ns
7 ns
8 ns

Figure 34 The ModelSim waveform using the dynamic synchronization mode

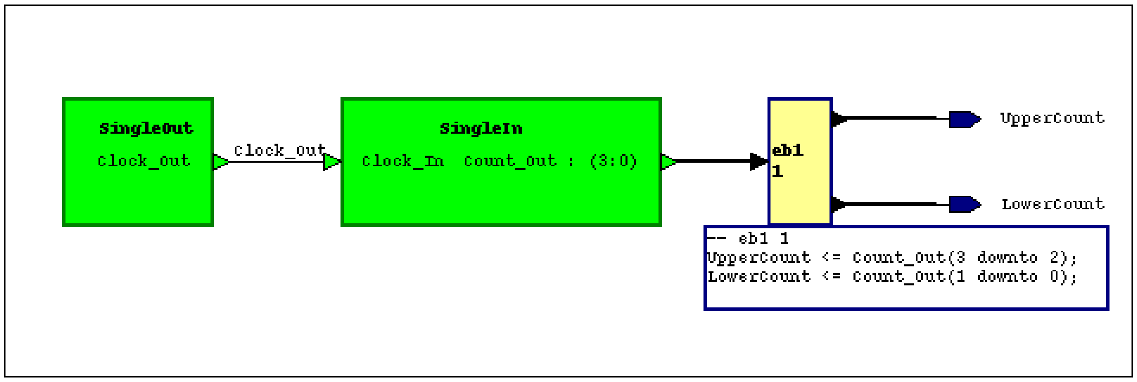
In both of the waveforms in Figure 33 and 34, we see a snapshot between 1ns and 8ns. In Chatoyant, we see the transitioning of the clock with a period of 5ns or 200MHz. In ModelSim, we see the clock from Chatoyant as the top most signal trace. Closer examination shows a brief logic transition from 0 to 'X' to '1' around 2.5ns. The same is seen at 5ns with a transition from 1 to 'X' to 0. The 'X' state is identified by a discontinuity in the transition of the signal trace. This is due to the brief period in time when the signal transitions between the threshold level of the logic, and thus is in an unknown state. We see this behavior for this particular case since there is an intermediate event from when the piece-wise linear source transitions from the original state of a signal to the new state. This is defined as the mid-point for the voltage level sent through the interface. This proves that the interface is properly handling the different voltage regions when converting from an electrical signal to a standard logic signal for the HDL design.

This system provided a first look into the performance of the co-simulation interface based on Chatoyant being the primary provider of events. However, this system accounts for only one bit of execution, as did the system presented in Section 5.2. The following section now looks at a larger example with multi-bit buses crossing the interface, first from the standpoint of ModelSim being the primary supplier of events.

5.4 MULTI-BIT, MODELSIM TO CHATOYANT APPLICATION

5.4.1 System Description

The system described in this section looks at the impact on performance when considering the use of buses. We will first look at the situation when ModelSim is the primary source of events. This is accomplished by writing behavioral VHDL code to create a 4-bit counter. Figure 35 shows the graphical view of this block diagram in the HDL Designer program from Mentor Graphics.



```
-- eb1 1
UpperCount <= Count_Out(3 downto 2);
LowerCount <= Count_Out(1 downto 0);
```

Figure 35 VHDL based block diagram for the multi-bit test system from ModelSim to Chatoyant

In Figure 35, we make use of the behavioral clock generator and counter made for the single-bit tests as the multi-bit count generator for this system. Two buses cross the interface. The first is the lower two bits of the count value and the second is the upper two bits of the count value. When these signals cross over the interface, they will be in the form of two, two-bit buses. This is done to stress the interface with two ports instead of only one.

The Chatoyant side of the interface contains the interface star with the two ports. Each port is then sliced into two separate single-bit paths to show the values of all bit lines. Figure 36 shows the Chatoyant palette with the interface star at the far left and all of the bit slices going to graphs.

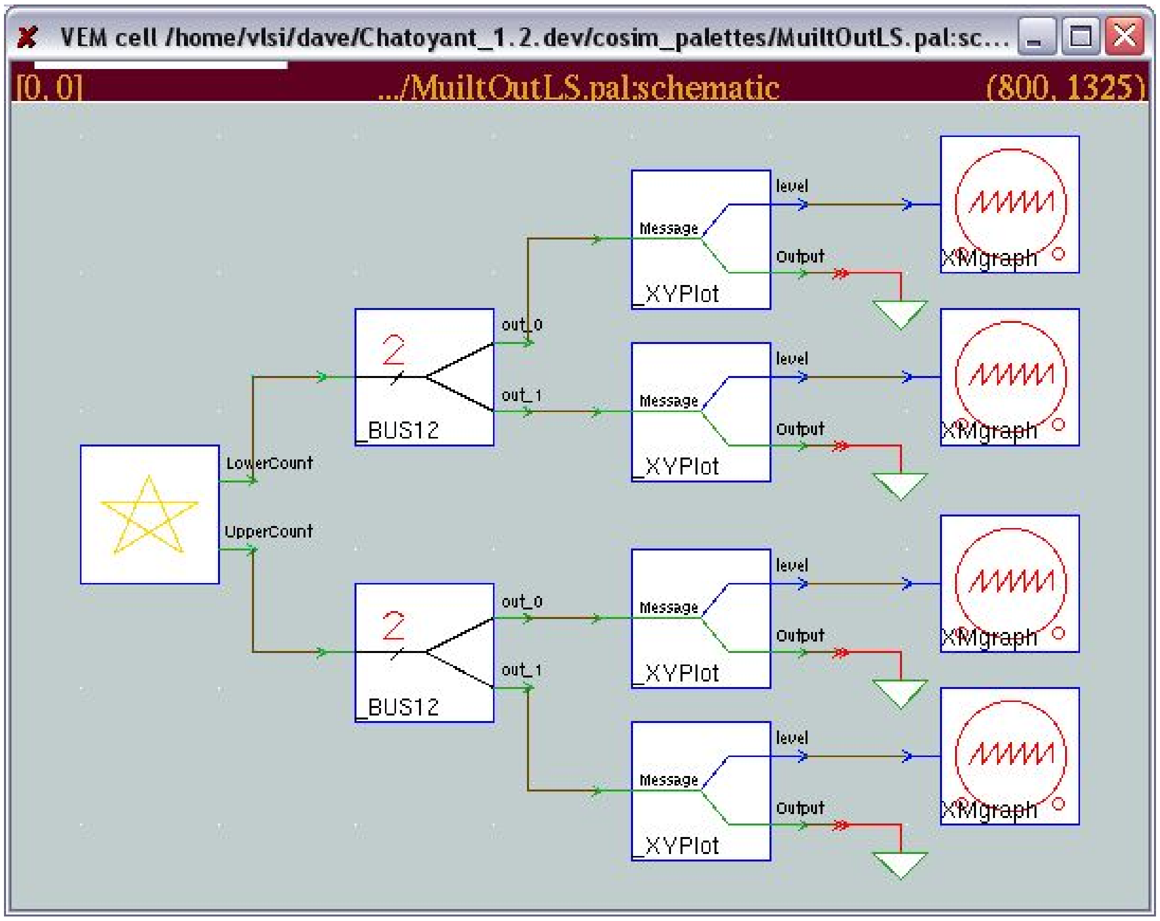


Figure 36 The Chatoyant palette for the multi-bit test from ModelSim to Chatoyant

Since this system is similar to the single-bit, ModelSim to Chatoyant system, we can predict that there will be approximately the same performance trend. We now present the results gathered from the simulations of the different modes of synchronization as well as the non co-simulation execution of the system within ModelSim.

5.4.2 Runtime Results and Analysis

We first provide the cross comparison graph of the events per second for the lock-step simulation at 1ps and 1ns resolution as well as the dynamic synchronization and non co-simulation runtimes. The summary of these results are found in Figure 37 and the detailed tables are available in Appendix D. For the non co-simulation bar, this is based on ModelSim's execution time, as was done in the single-bit, ModelSim to Chatoyant example.

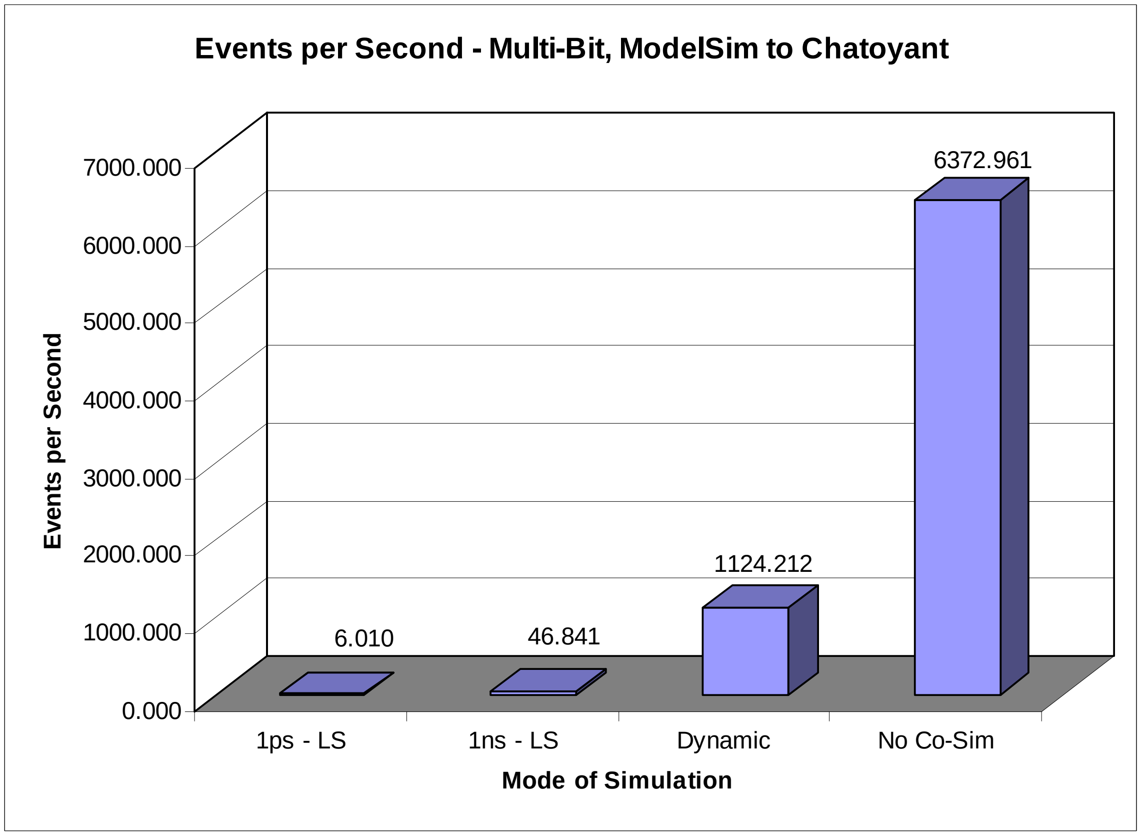


Figure 37 The events per second for the multi-bit, ModelSim to Chatoyant test system

We see in this graph that the non co-simulation execution is 5.66 times faster than the dynamic mode of synchronization. The greatest improvement in the co-simulation is shown between the dynamic simulation and the 1ps lock-step, with a 187.06 times in performance. This is due to the loss of events across the interface in the lock-step method (having only 252 events) versus no loss in the dynamic synchronization (having 379).

We see in the following code profiles, that the event processing is largest overhead when considering the dynamic synchronization co-simulation. The actual runtime of the interface based code is minimal compared to the event processing and other code execution. This additional overhead is associated with the null events produced from the synchronization as well as the fact that there are two events created for every transition port in the interface. This overhead distribution is seen in Figure 38.

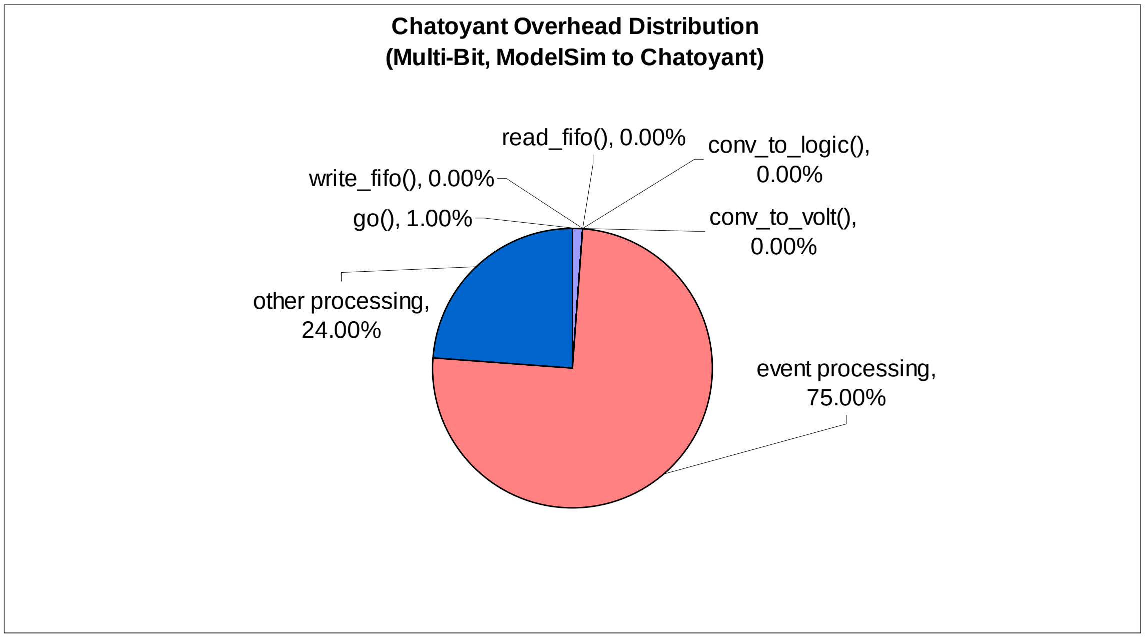


Figure 38 Chatoyant code profile for the multi-bit, ModelSim to Chatoyant system in dynamic mode

Lock-step processing distribution, seen in Appendix D, shows that over 66% of the execution time is spent within the synchronization function. This is accounted by the large quantity of null events created by the synchronization pulse. The lock-step's 66% compared to the dynamic synchronization's 1% execution contribution shows the reduction in the interfaces overhead when running in dynamic mode.

Figure 39 is the ModelSim code profile for the multi-bit, ModelSim to Chatoyant test system utilizing dynamic synchronization. The majority of the overhead for running this system is found in the *SyncInterface()* function within the interface code. The actual ModelSim processing overhead is minimal when compared to the interface overhead.

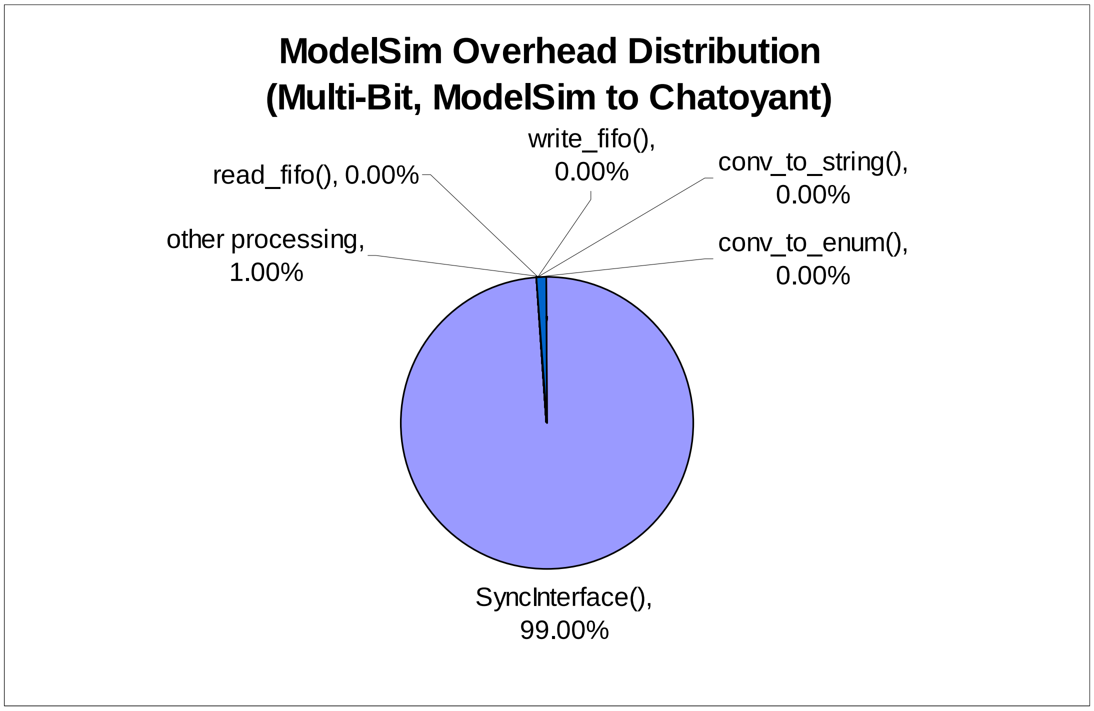


Figure 39 ModelSim code profile for the multi-bit, ModelSim to Chatoyant system in dynamic mode

The final results to examine are the waveforms. In Figures 40 and 41 we show the Chatoyant and ModelSim waveforms, respectively. The ModelSim waveform has one additional signal, the design's clock, shown for reference purposes.

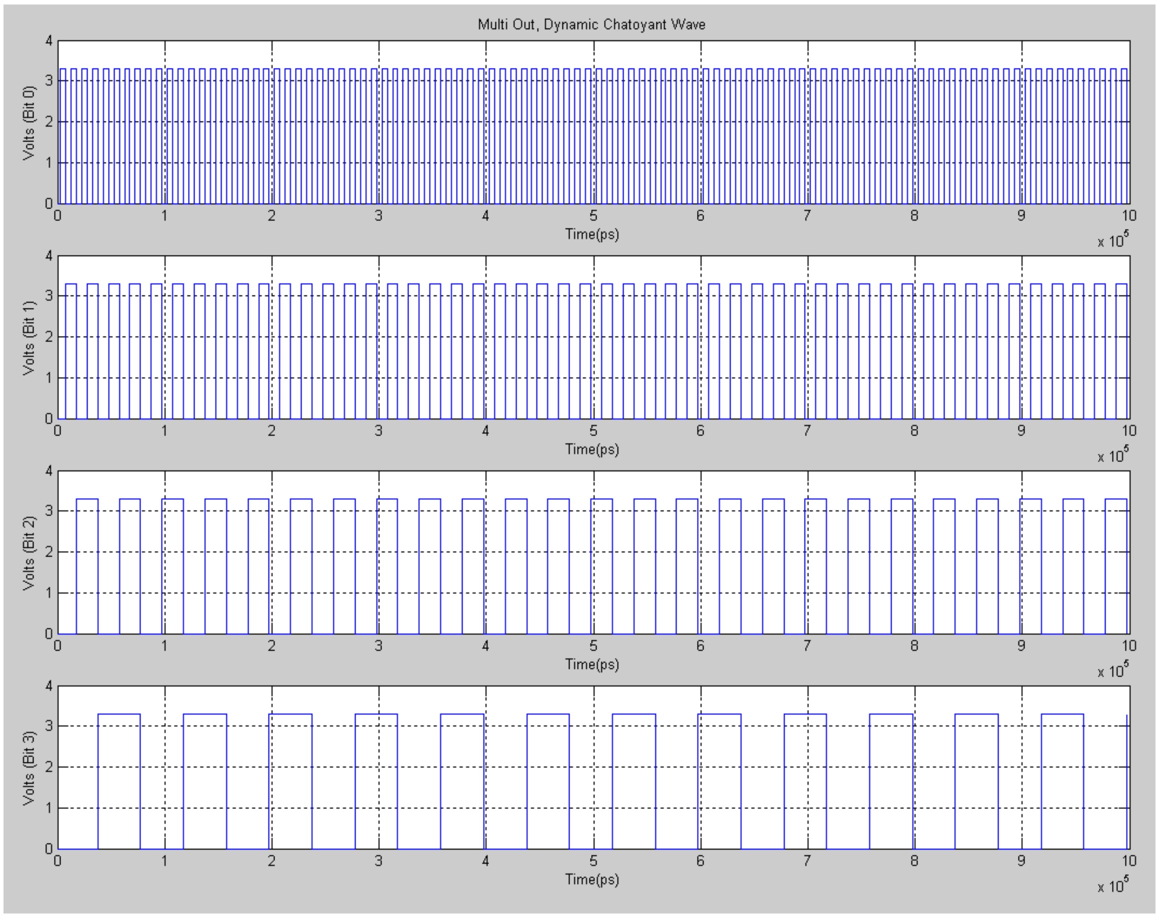


Figure 40 Chatoyant waveform for the multi-bit, ModelSim to Chatoyant system in dynamic mode

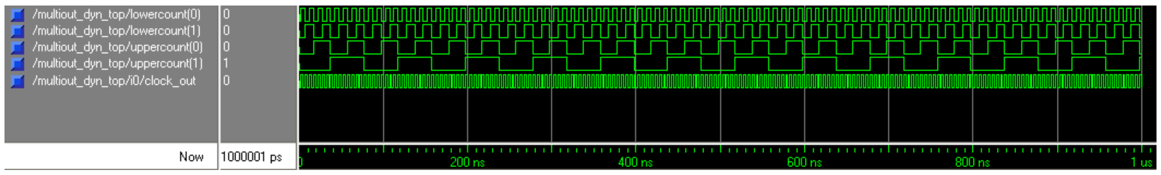

/multiout_dyn_top/lowercount(0)
/multiout_dyn_top/lowercount(1)
/multiout_dyn_top/uppercount(0)
/multiout_dyn_top/uppercount(1)
/multiout_dyn_top/i0/clock_out
Now
1000001 ps
200 ns
400 ns
600 ns
800 ns
1 us

Figure 41 ModelSim waveform for the multi-bit, ModelSim to Chatoyant system in dynamic mode

A closer inspection of these graphs shows that accuracy is maintained. Each bit toggles at different frequency. The signal group with the highest toggle rate produces the most events and thus incurs the most synchronization as well as event processing overhead in the simulators as well as the interface.

We will now examine the opposite case to the system described in this section. This will now make Chatoyant the source of the events, with multiple bits, and ModelSim the destination of these events.

5.5 MULTI-BIT, CHATOYANT TO MODELSIM APPLICATION

5.5.1 System Description

This section examines the case of a multi-bit bus originating in Chatoyant and being used in ModelSim. This makes Chatoyant the primary source of all events and ModelSim will simply use these events for some trivial purpose.

As seen in the single-bit, Chatoyant to ModelSim system in Section 5.3, we utilize the piece-wise linear source star for creating the events. In this case we have 4 piece-wise linear source stars, one for each bit in the bus. These bits are then paired off into a high order pair and a low order pair, to create a pair of two bit buses. These buses then traverse the interface into ModelSim via the Chatoyant interface star created for this test system. Figure 42 shows a screen capture for the Chatoyant palette created for this system. The far left of the diagram shows the piece-wise linear source stars, followed by graphs used to display each bit, with each slice being pared and merged into a two bit bus, and then entering the interface star.

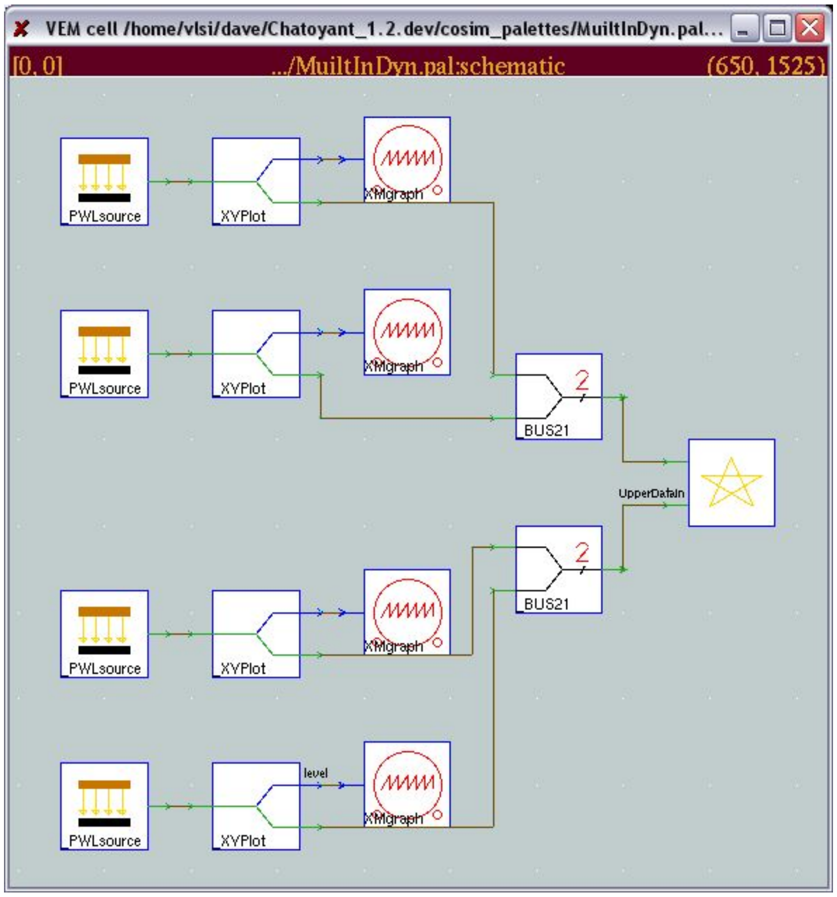
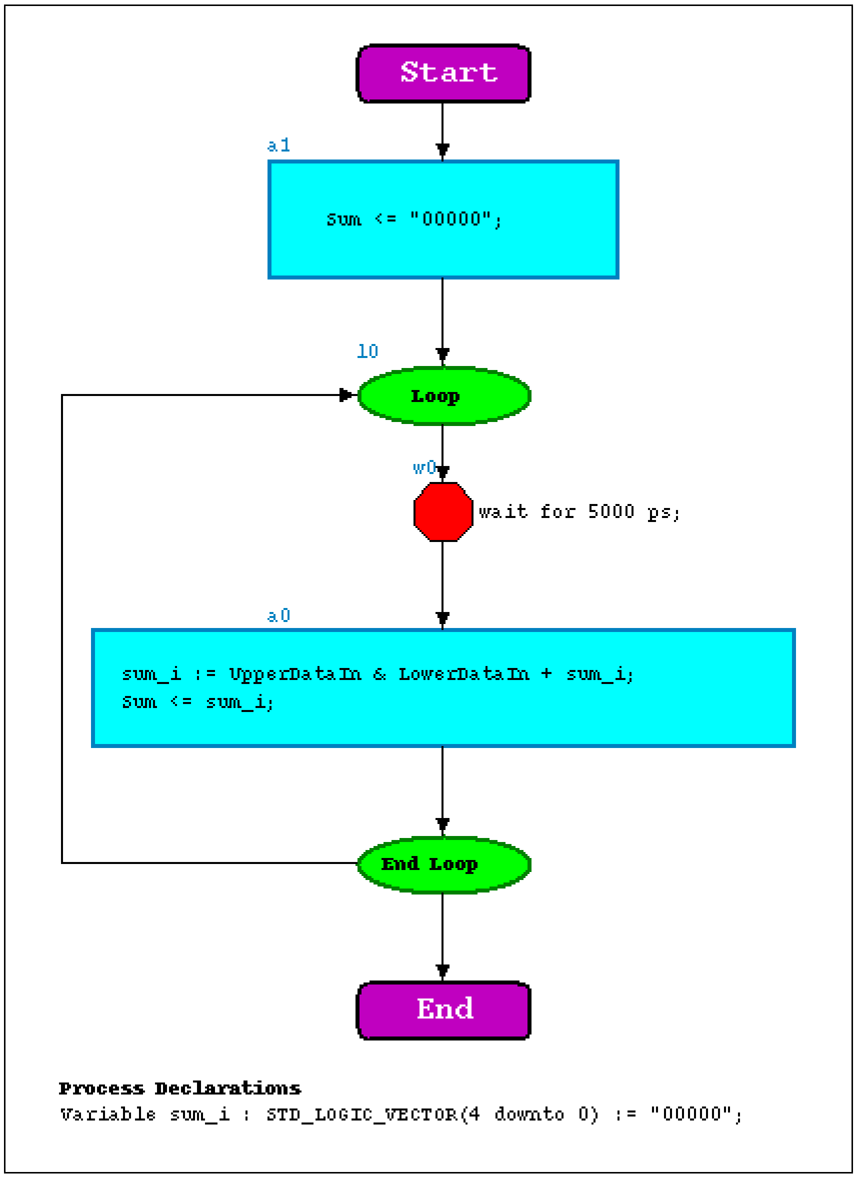

VEM cell /home/vlsi/dave/Chatoyant_1.2.dev/cosim_palettes/MuiltInDyn.pal...
[0, 0]
.../MuiltInDyn.pal:schematic
(650, 1525)
PWLsource
XYPlot
XMgraph
BUS21
2
UpperDataIn
level

Figure 42 Chatoyant palette for the multi-bit, Chatoyant to ModelSim test system

Within ModelSim, the buses are used to create a running sum between each of the two bit buses and the previous sum value. This design is constructed through behavioral VHDL. This code is represented as a flow chart diagram within Mentor Graphics' HDL Designer tool in Figure 43. This diagram shows a forever loop that first waits for 5ns and then performs the sum between the upper two bit bus, the lower two bit bus, and the previous sum value. The sum is initially set to all zeros. This loop will run until the actual simulation stops within ModelSim.



```
Sum <= "00000";
```

```
sum_i := UpperDataIn & LowerDataIn + sum_i;
Sum <= sum_i;
```

```
Process Declarations
Variable sum_i : STD_LOGIC_VECTOR(4 downto 0) := "00000";
```

Figure 43 ModelSim flowchart view for the multi-bit, Chatoyant to ModelSim test system

We now examine the results yielded from the different simulations performed with this system to gain insight on how it performs compared to the non co-simulation execution within Chatoyant.

5.5.2 Runtime Results and Analysis

We begin our analysis of this test system by examining the events per second for the different modes of co-simulation versus the non co-simulation mode. The non co-simulation execution is measured from the Chatoyant system in Figure 42 executing without the ModelSim Star. Seen in Figure 44 is the graph of the different events per second measurements for these modes.

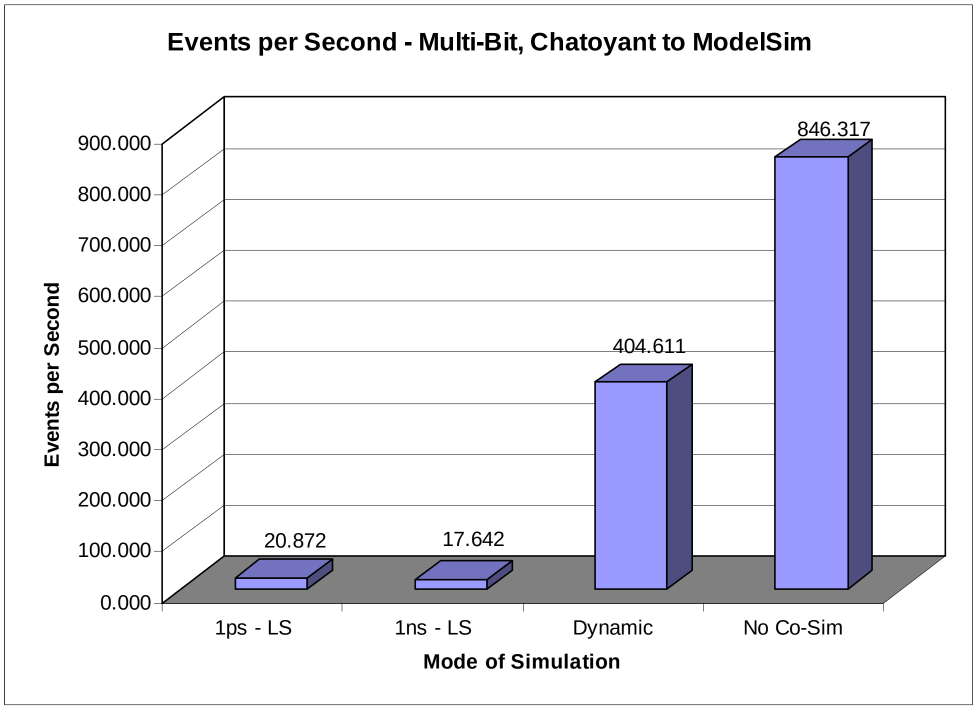


Figure 44 Events per second results for the multi-bit, Chatoyant to ModelSim test system

As shown in figure 44, the non co-simulation mode of execution shows only a 2.1 times speedup in event throughput. Meanwhile, the dynamic mode of co-simulation provides a 19.4 times speedup over the lock-step co-simulations, while producing an order of magnitude in terms of the number of events traversing the interface, than seen in either lock-step resolution.

This system provides results that do show a speedup between dynamic and lock-step modes of simulation. The 1ns lock-step mode is seen to be slower than the 1ps lock-step due to the fact that few events are traversing the interface (because of detail loss) in nearly the same time, show in Appendix E. When comparing to the 1ps lock-step co-simulation, the dynamic mode is 19.4 times faster since the dynamic co-simulation produces a higher event throughput in the interface.

We continue the examination of this system by analyzing the code profiles. Figure 45 shows the profile for Chatoyant. Here we see that over half of the execution time in Chatoyant is spent on other processing tasks. 32.11% of the time is spent processing events and 10.98% is spent in the interface star. The larger quantity of execution time spent in other Chatoyant processing functions is most likely due to the four piece-wise linear source stars and associated graphs. Since the interface handles the multi-bit buses as one port instead of individual signals, there is not as much processing time needed to handle event changes.

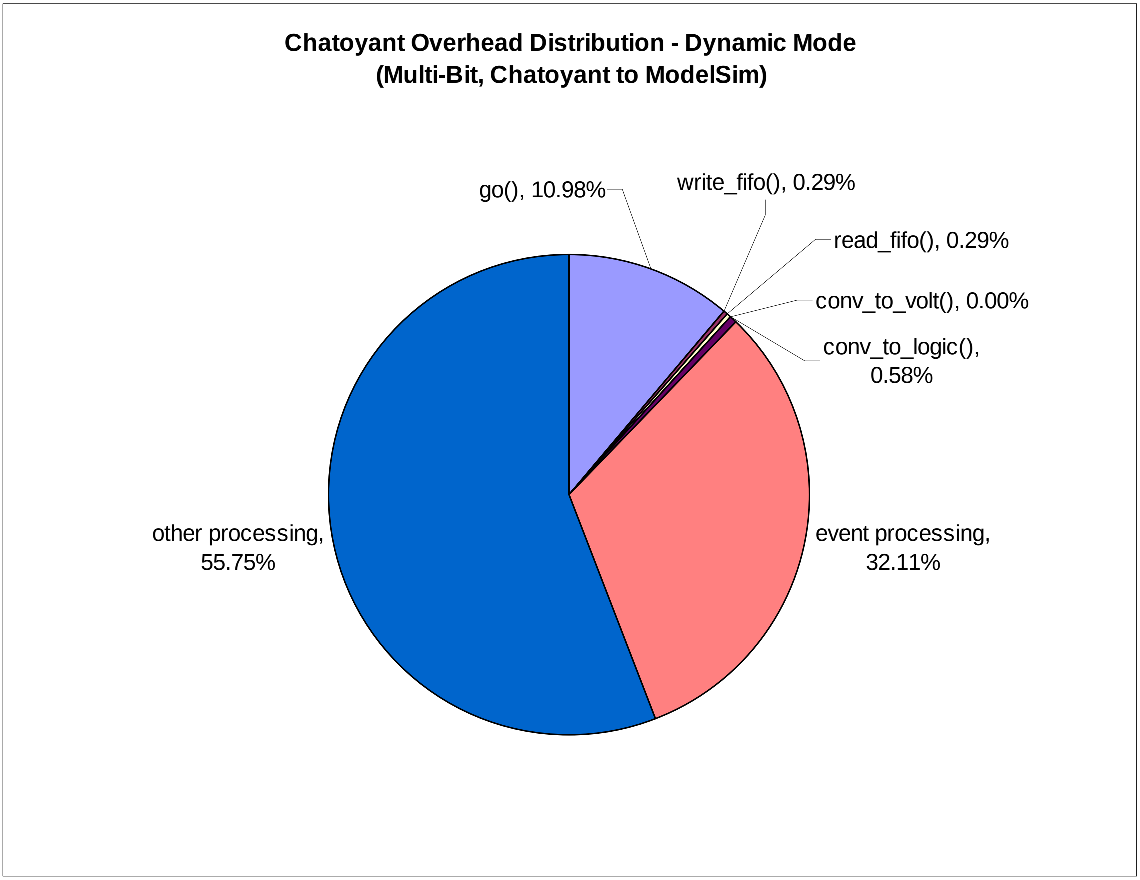


Figure 45 Chatoyant profile for the multi-bit, Chatoyant to ModelSim system in dynamic mode

The ModelSim profile, seen in Figure 46, shows a smaller overhead in terms of the interface code compared to the previous systems. In the case of this system, the *SyncInterface()* function is still the highest percentage of the runtime, but is closer to the 38% of all other ModelSim processing. The other functions used by the interface account for around 18% of the total execution. This tighter distribution of the function calls is most likely due to the higher number of events and efficiency of the interface in handling them, leaving more processing to be done in the ModelSim kernel.

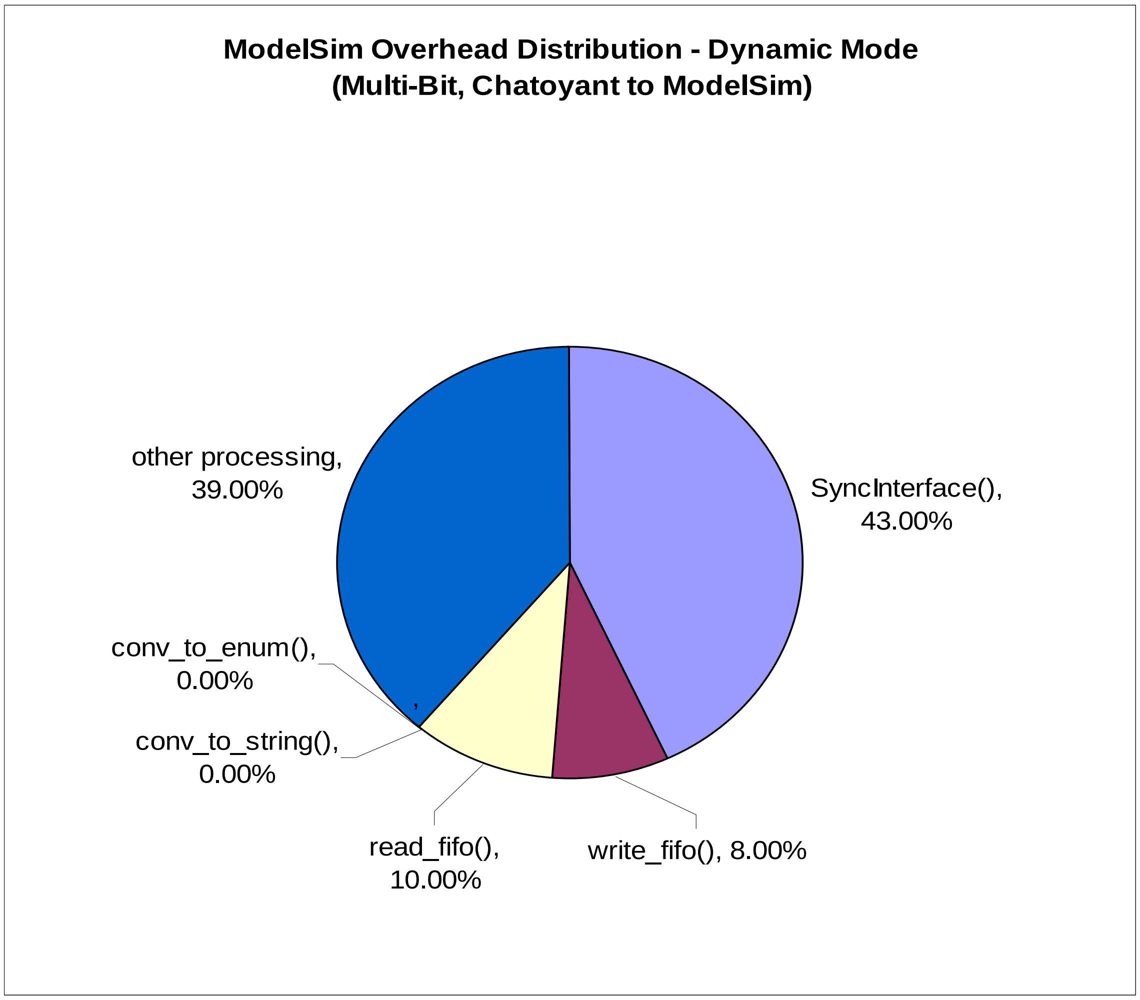


Figure 46 ModelSim code profile for the multi-bit, Chatoyant to ModelSim system in dynamic mode

The final examination of this system is the analysis of the waveforms produced from the simulations. We focus our analysis on the dynamic co-simulation since it has provided the best performance for the interface. Figure 47 shows the Chatoyant graphs for each bit. These bits toggle in periodic fashion, each bit being the inverse of the other within a bus pair.

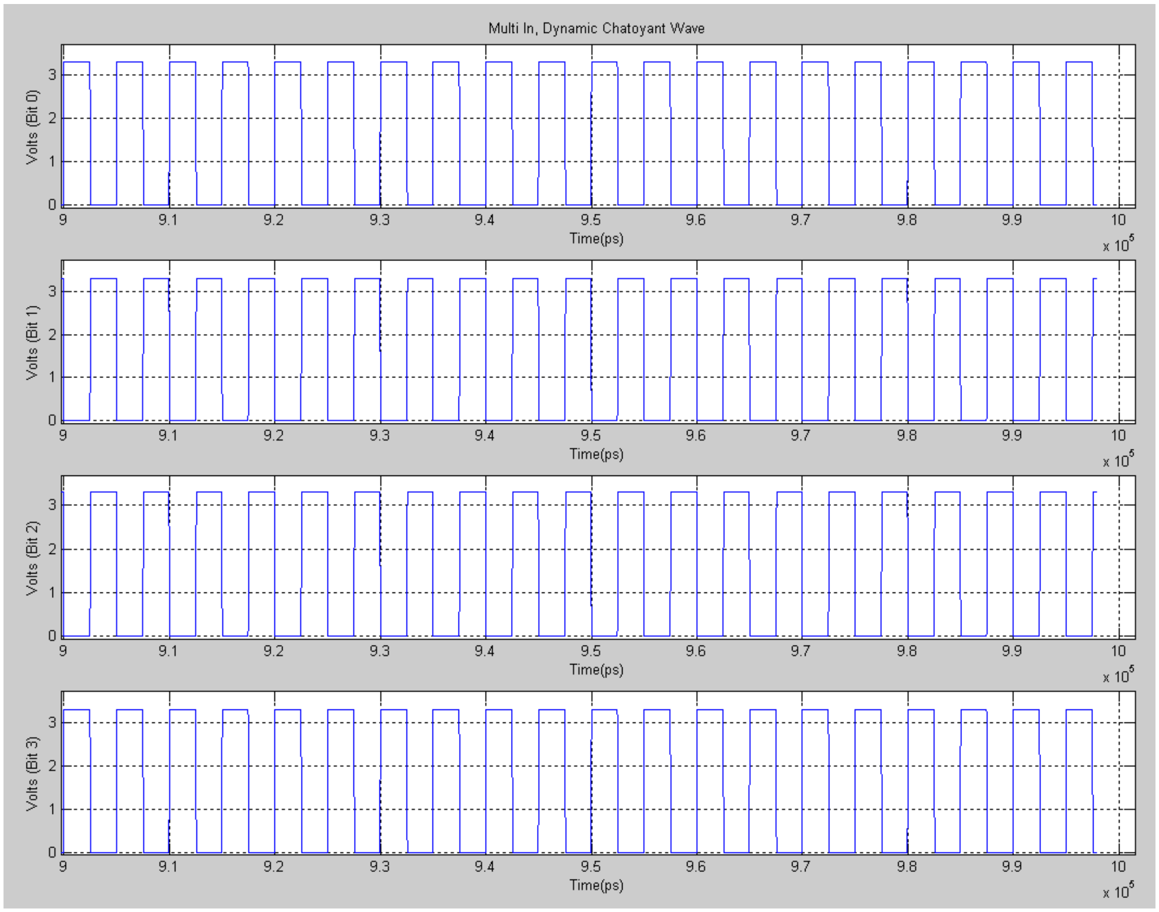


Figure 47 Chatoyant waveform for the multi-bit, Chatoyant to ModelSim test system

In Figure 48, we see the resulting ModelSim waveform. Here we see the same pattern as produced by Chatoyant, with the addition of the unknown crossover produced by the bits toggling in and out of the unknown voltage region. The sum produced by the behavioral VHDL code is also shown with the incoming signals as a reference, shown as the top most signal.

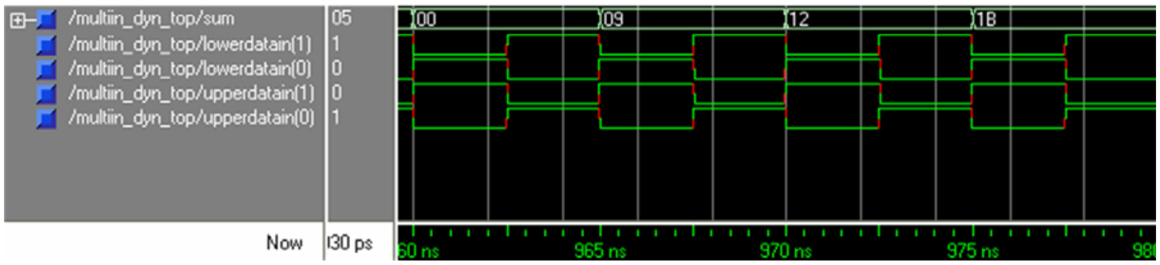

/multiin_dyn_top/sum
/multiin_dyn_top/lowerdatain(1)
/multiin_dyn_top/lowerdatain(0)
/multiin_dyn_top/upperdatain(1)
/multiin_dyn_top/upperdatain(0)
05
1
0
0
1
00
09
12
18
Now
130 ps
60 ns
965 ns
970 ns
975 ns
98

Figure 48 ModelSim waveform for the multi-bit, Chatoyant to ModelSim test system

With the systems described and analyzed in this section, we conclude the examination of the four primary test cases. For the complete set of runtime results see Appendix E. We now examine two larger systems that represent real-world designs that are currently fabricated and being tested. The first system discussed is the Smart Pixel Optical Transceiver system created at the University of Delaware.

5.6 SMART PIXEL OPTICAL TRANSCEIVER – SPOT

5.6.1 System Description

The system presented in this section represents a real-world design created at the University of Delaware. The Smart Pixel Optical Transceiver, or SPOT, system is a prototype design used to prove the feasibility of an optics based high speed communication interconnect (40). Using a digital design to generate 8-bit parallel data, two data channels and one high speed clock signal are serialized and transmitted through analog circuitry to Vertical Cavity Surface Emitting Lasers, or VCSEL's, arrays to a series of lenses and received by a matching set of photodetectors. The receivers translate the optical signal into a series of electrical signals, which in turn are amplified through analog circuits and then sent through digital circuitry which recovers the serialized data and translates it into a pair of 8-bit parallelized words. Each serialized data signal is a Double-Data Rate channel and is sent with a serialized clock, which is a clock that is 4X faster than the original user clock. This clock speedup is accomplished through a Phased-Lock-Loop, or PLL, within a Xilinx Virtex Field Programmable Gate Array, or FPGA, chip. Each channel is sent as parallel data along with the 4X clock to a transceiver chip which accomplishes the serialization/de-serialization (SERDES) as well as transmission and reception of the optical signals. Figure 49 shows the actual prototype system.

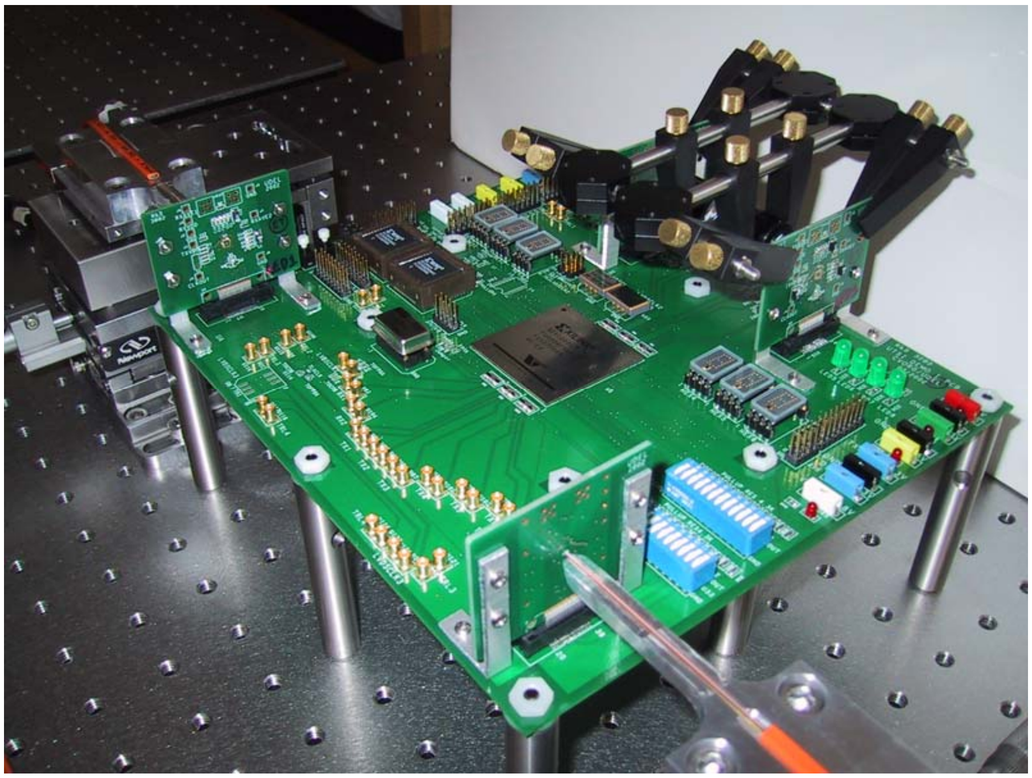


Figure 49 SPOT prototype test system [(40)]

The primary user clock in this system is a 125MHz clock, which equates to the data being transferred through the optics at 500Mbs, or 4X the clock. The data being sent is in two independent channels, each being a simple 8-bit count value, with one sending an up-count and the other sending a down-count value. Figure 50 shows the top level block diagram for the digital side of the SPOT system, created in Mentor Graphics' HDL Designer. In this diagram, we see the block diagram representing the code found within the Virtex FPGA (left), the transmitter logic found within the SPOT transceiver (upper right), and the block containing behavioral VHDL code for the SPOT receiver (lower right).

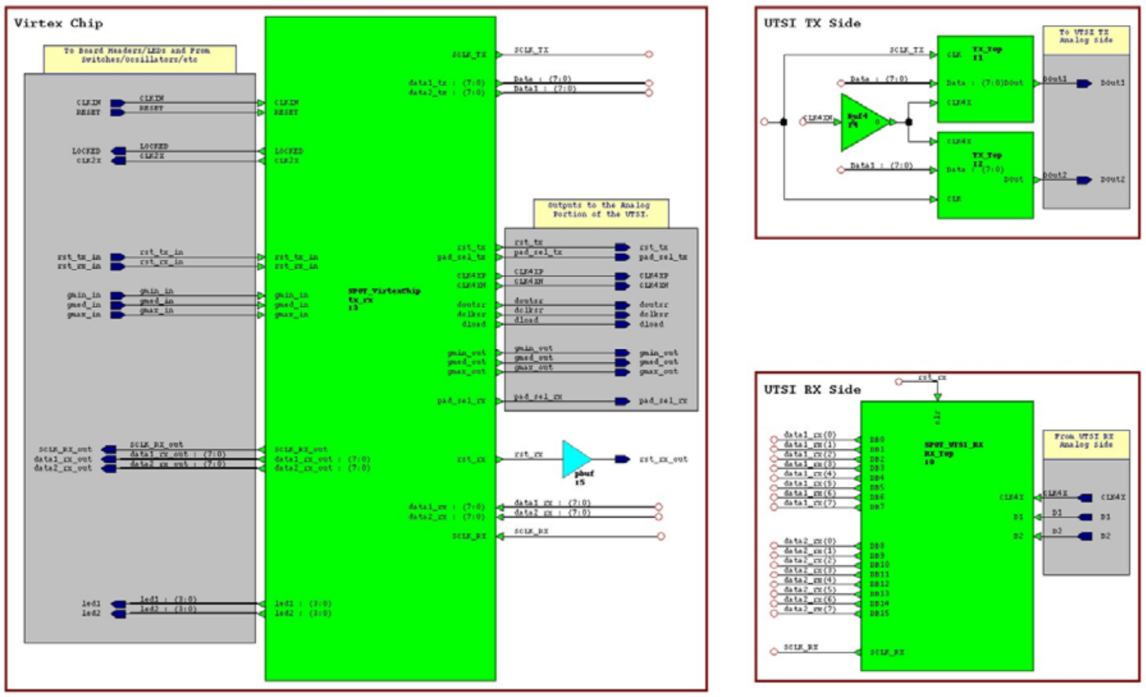


Figure 50 SPOT top-level VHDL block diagram

The Chatoyant galaxy for this system is simple for the purpose of collecting and profiling the performance of the co-simulation interface. A full system simulation would contain the piece-wise linear circuit solver along with the VCSEL model star for each clock and data channel with its corresponding receiver and analog amplifier circuits. This system was tested in the simplified form to better understand the impact of the interface by itself. Figure 51 shows the simplified Chatoyant palette with the interface star in the center, a set of graphs from each signal, and then a set of zero-delay interrupters, provided by Ptolemy, indicated as diamonds on the signal path. This setup causes a loop-back operation during simulation so that whatever is sent from ModelSim will be received at a few time steps later, defined by the rise/fall time constant.

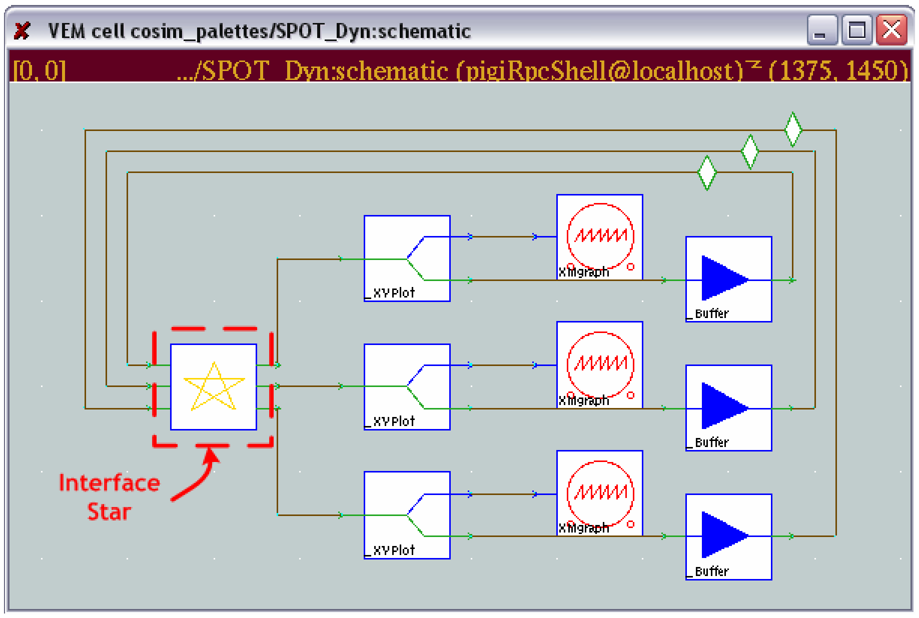

VEM cell cosim_palettes/SPOT_Dyn:schematic
[0, 0]
.../SPOT_Dyn:schematic (pigiRpcShell@localhost)^z (1375, 1450)
XYPlot
Xmgraph
Buffer
Interface Star

Figure 51 Chatoyant palette example for the SPOT system

We now investigate the runtime performance of this system as we did in the previous test systems. Since this system ultimately has ModelSim as the initial source of events due to the data and clock channels being driven by the FPGA and transmitter VHDL code, we should see results that follow closely to the single and multi-bit ModelSim to Chatoyant systems.

5.6.2 Runtime Results and Analysis

Due to the amount of data/clock information being transmitted in this system, we expect to see a reduction in the runtime of the simulation in both the co-simulation cases as well as the non co-simulation case. This is mostly due to the larger design present in ModelSim and the impact this has on the runtime capability of ModelSim. Figure 52 shows the comparison between the different simulation modes. The non-co-simulation system results are based on the simulation executing only in ModelSim since it is the initial source of events.

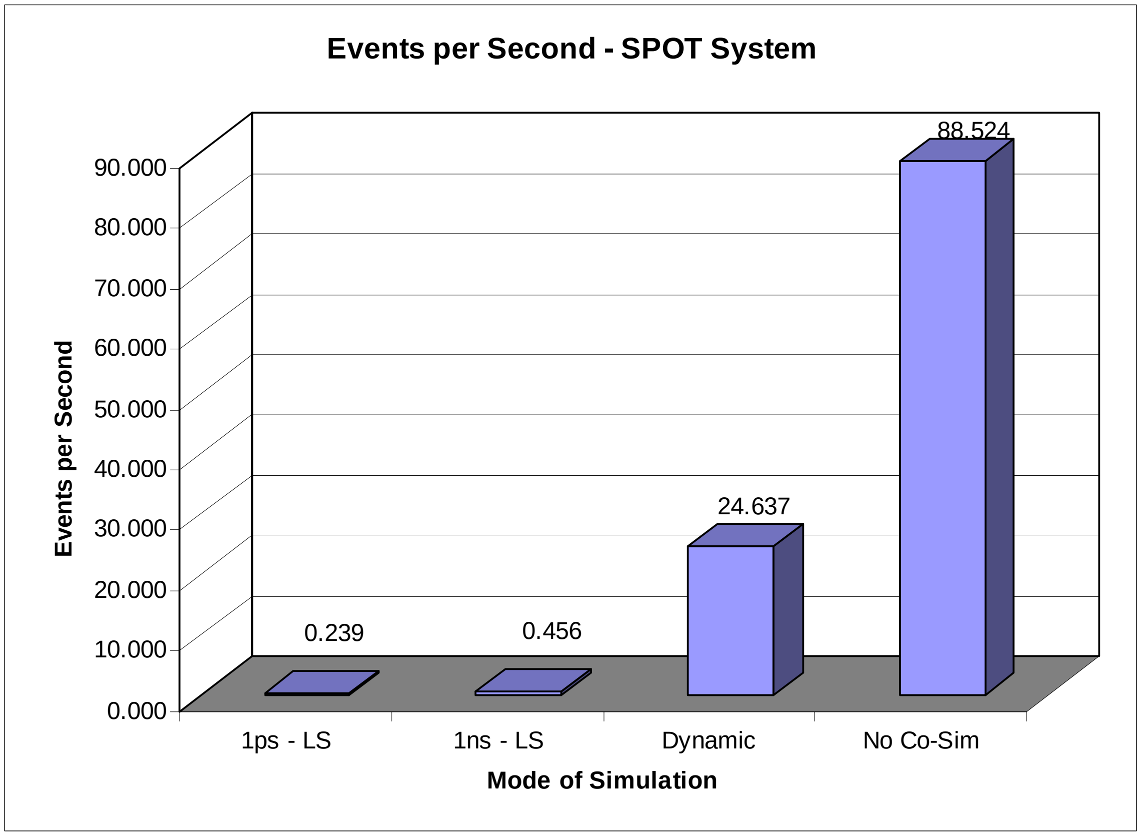


Figure 52 Events per second measurements for the different simulation modes for the SPOT system

As we see in Figure 52, there is a reduction in the events per second compared to the previous systems. The non co-simulation system is only 3.49 times faster over the dynamic co-simulation mode. The dynamic co-simulation mode has a speedup of 54 times over the 1ns lock-step mode and a 103 times speedup over the 1ps lock-step mode. The lock-step system does not see a loss of event accuracy between the 1ps to 1ns resolutions, mostly due to the even clock period being at every 8ns with the half period at 4ns. However, event though the accuracy is maintained in the lock-step mode, the dynamic synchronization method proves to be the optimal mode for co-simulation.

When considering the impact of the overhead in ModelSim as well as Chatoyant, we examine the code profiles provided by both simulators. Figure 53 provides a graph of the Chatoyant execution distribution. We see in this graph that 43.72% of the time spent during simulation is found within the event processing methods. The *go()* code of the interface star is the second highest consumer of execution time. The other processing found in Chatoyant is limited to only 10.97%, which is explained by the fact that ModelSim is the primary source of events and that there is little computation performed in Chatoyant.

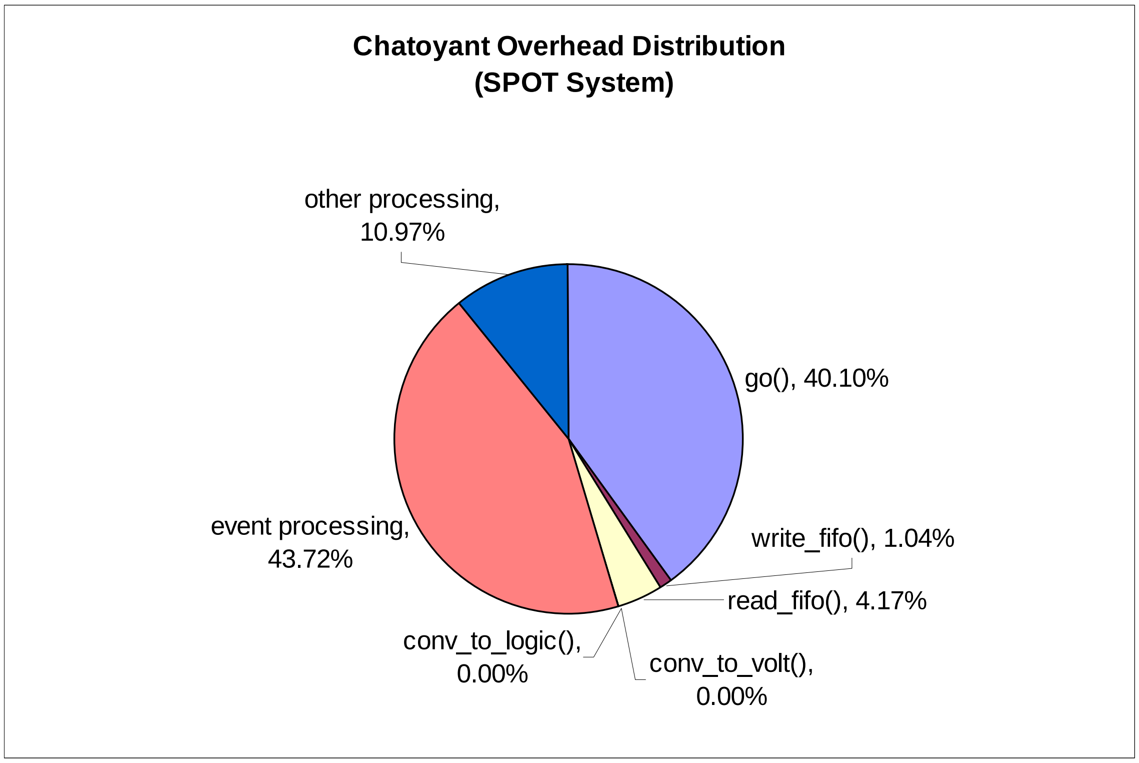


Figure 53 Chatoyant code profile for the SPOT system in dynamic mode

The ModelSim code profile is seen in Figure 54. Here we see that the majority of the code execution is found within the *SyncInterface()* function, as expected. However, the remaining ModelSim execution does consume 18% of the execution which is higher than most of the test systems. This can be explained by the addition simulation computation needed to handle the larger design and gate-level constructs.

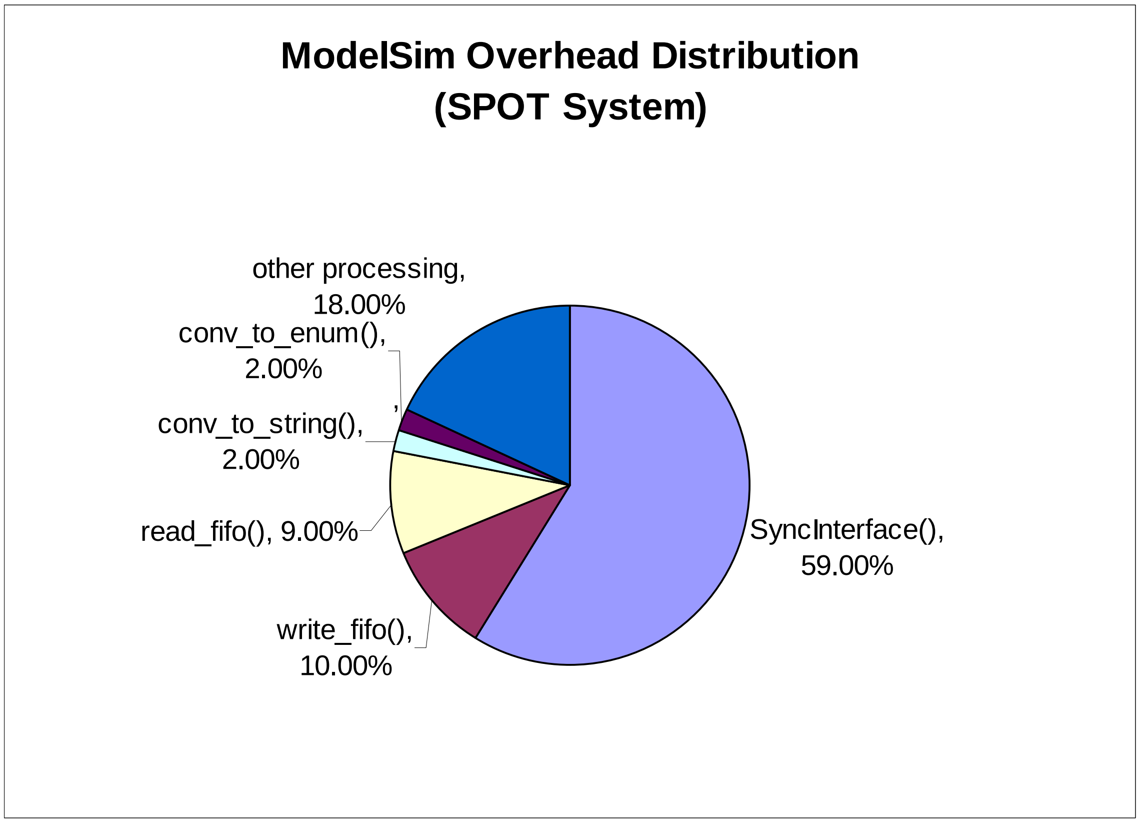


Figure 54 ModelSim code profile for the SPOT system in dynamic mode

We conclude our analysis of the SPOT system runtime data by looking at the resulting output waveforms from both ModelSim and Chatoyant. We see in Figure 55 the waveform from ModelSim. Here we see the primary user clock running at 125MHz and then the serialized data/clock channels being sent to Chatoyant via the co-simulation interface. The signals are then received back from Chatoyant into ModelSim with a few pico-seconds of delay (due to rise/fall time skewing). The bottom of the waveform shows the recovered 8-bit data channels which show the count-up and count-down values that were originally serialized on the transmitter side.

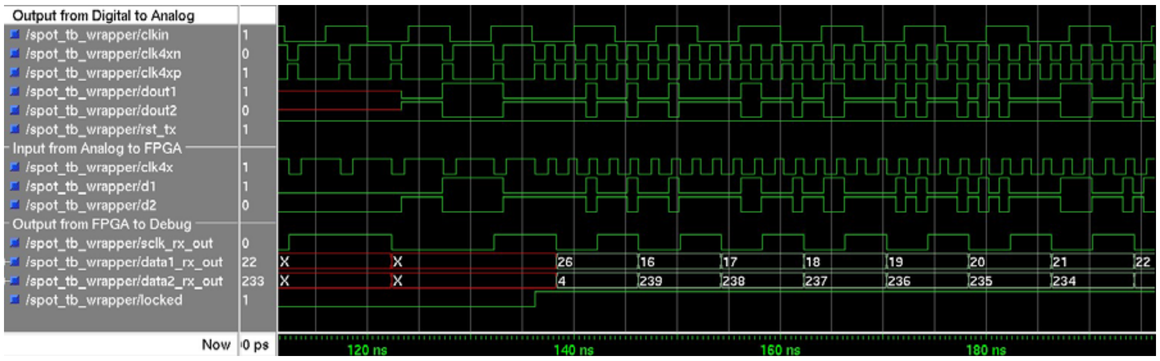


Figure 55 ModelSim waveform for the SPOT test system in dynamic mode

The Chatoyant results are split into three waveforms. The 4X clock is seen in Figure 56, the first data channel is seen in Figure 57, and the second data channel is seen in Figure 58. These are identical waveforms as seen in the ModelSim waveform from the output of the transceiver logic.

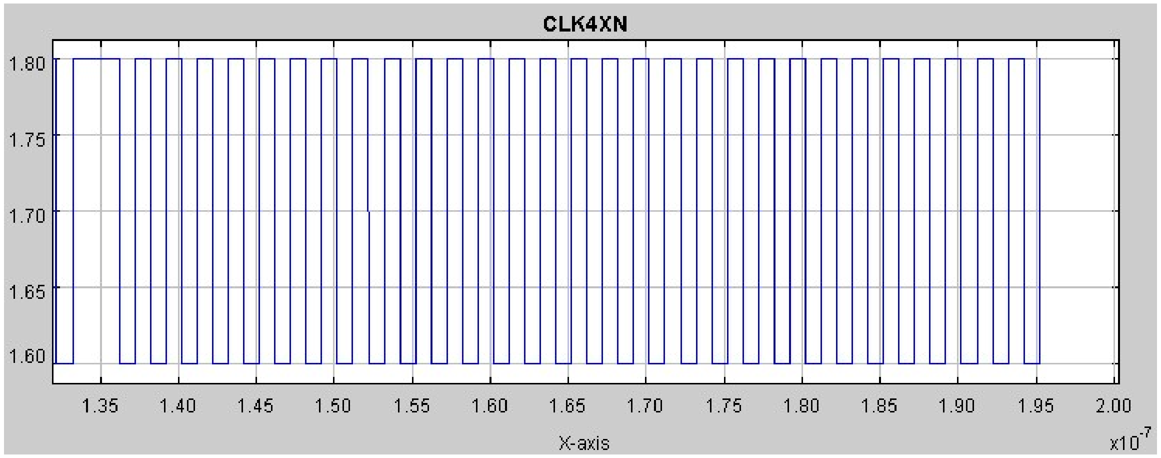


Figure 56 Chatoyant waveform for the 4X clock for the SPOT system in dynamic mode

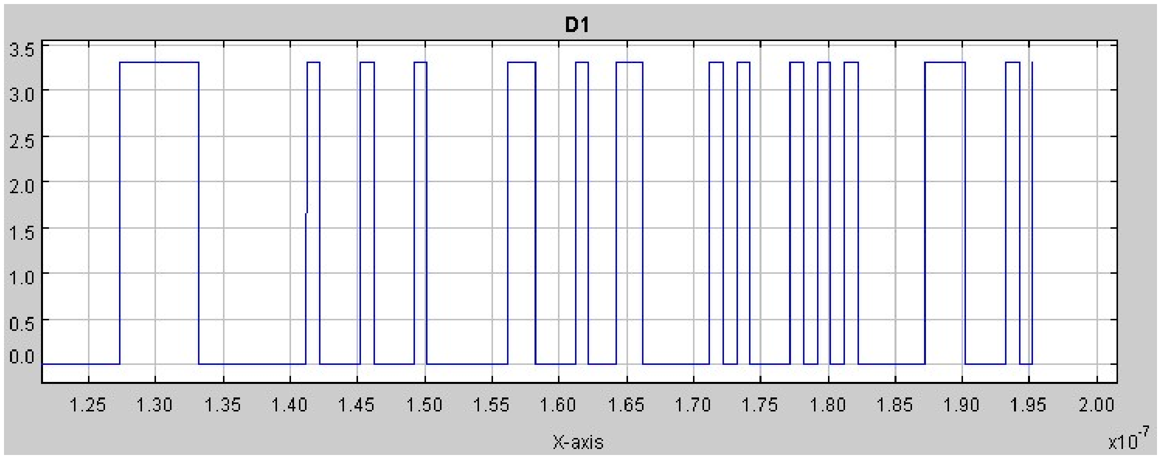


Figure 57 Chatoyant waveform for the first data channel for the SPOT system in dynamic mode

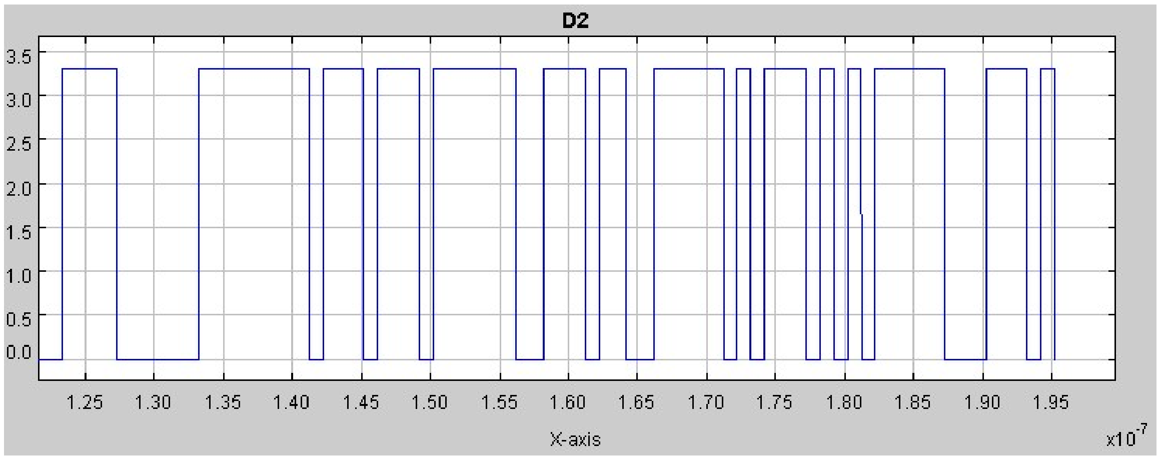


Figure 58 Chatoyant waveform for the second data channel for the SPOT system in dynamic mode

In this section we saw the first of two real-world applications using the co-simulation interface. Appendix F provides the detailed runtime results for this system. We saw that the dynamic co-simulation mode provided a significant speedup over the lock-step mode. We now discuss the last system tested using the co-simulation interface in the following section.

5.7 FIBER IMAGE GUIDE - FIG

5.7.1 System Description

The final system tested in this thesis is by far the largest in terms of functional blocks in both VHDL and Chatoyant. This system is the Fiber Image Guide, or FIG, design, created at the University of Pittsburgh (41). FIG, as a complete system, is an 8x8 crossbar switch. There are 8 channels that are 8-bits each. This system consists of 64 bits of optical input into the first stage of the crossbar. These 64 bits are recovered from the optics into amplified digital signals. The bits then enter the first of three digital crossbar switching logic stages. Based on configuration logic, the bits are then switched to the proper transmission channel to a set of analog circuits that drive a large array of VCSEL's. The fiber image guide directs the laser light of each VCSEL transmitter to a corresponding photodetector of an identical crossbar chip. The process is then repeated again for another set of bit translations, based on more control information. At the third chip, the final crossover occurs and the output of the chip is another set of external fiber bundles with the fully switched bit data. To illustrate this, refer to Figure 59 which depicts the top and bottom of the complete FIG multi-chip module.

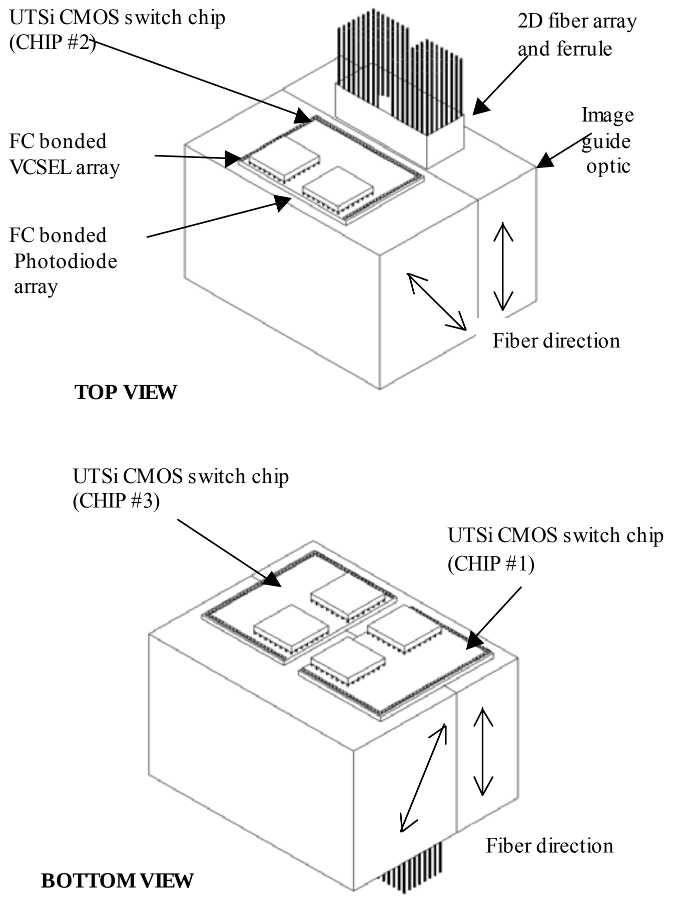

UTSi CMOS switch chip (CHIP #2)
2D fiber array and ferrule
Image guide optic
FC bonded VCSEL array
FC bonded Photodiode array
Fiber direction
TOP VIEW
UTSi CMOS switch chip (CHIP #3)
UTSi CMOS switch chip (CHIP #1)
Fiber direction
BOTTOM VIEW

Figure 59 The FIG MCM architecture (42)

The complete FIG system is massive and would require a potentially long simulation runtime due to the amount of optical, analog, and digital components (essentially 64 individual paths times 3 stages or 192 major components). To reduce the simulation, we examine a small segment of the design, namely the small segment of the incoming fiber optics for two bits of the first two buses through the digital component, and then out to a set of a data sinks (black holes in Ptolemy). This reduction in the simulation performed is done for the same reasons as was seen in the SPOT system. This reduction allows us to perform a simulation that examines the co-simulation interface performance.

In terms of the Chatoyant palette, there are a set of piece-wise linear sources that represent the incoming optical/analog data bits for any given stage in the crossbar. These bits are merged into buses in a similar fashion as in the multi-bit, Chatoyant to ModelSim system. These buses are then sent through the interface to VHDL code that models the crossbar control logic. The result of this logic is then fed as two buses from the interface star into two graphs for monitoring. In the real system simulation, these buses would feed analog circuits that would in turn drive the VCSEL array. We see this system in Figure 60.

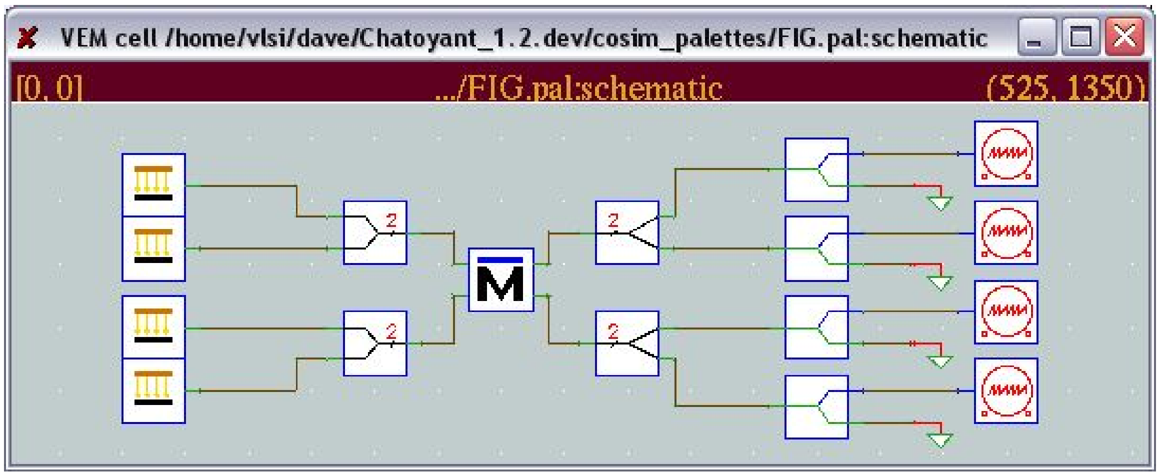

VEM cell /home/vlsi/dave/Chatoyant_1.2.dev/cosim_palettes/FIG.pal:schematic
[0, 0]
.../FIG.pal:schematic
(525, 1350)
2
2
M
2
2

Figure 60 Chatoyant palette for the FIG co-simulation test

The digital logic for the crossbar control component within FIG is illustrated in Figure 61. In this diagram we see the top-level block diagram for the control word cache with the main controller (top left blocks) and the main multiplexer units (right side). The HDL of this design is primarily gate-level code with the control logic coded in structural Register Transfer Level (RTL) VHDL. This has an impact on the performance of the digital logic design in ModelSim, seen in the following subsection.

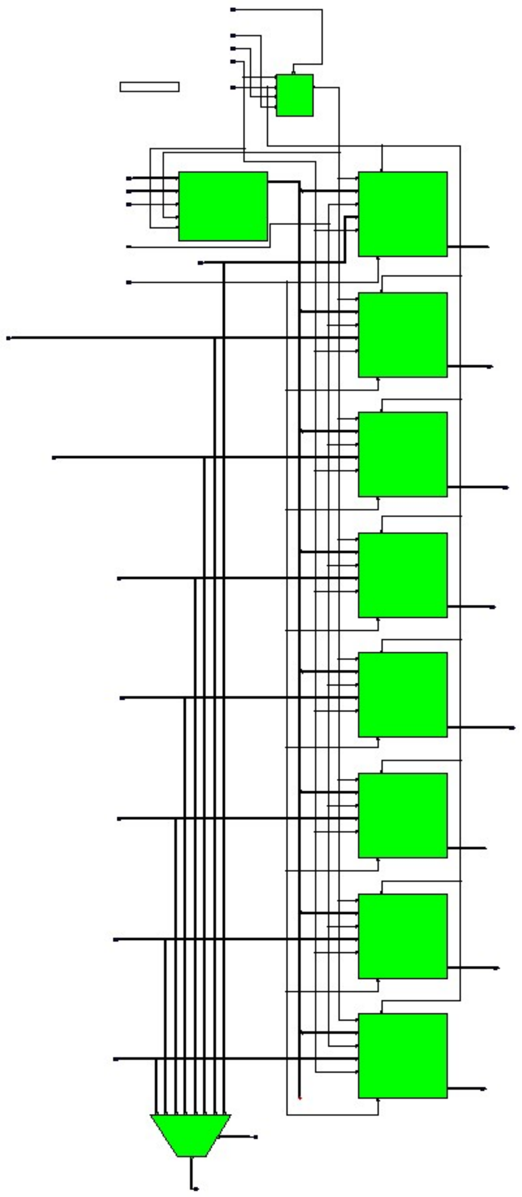


Figure 61 VHDL diagram of the FIG crossbar chip (41, 42)

We now examine the runtime results gathered from the different simulation modes for the FIG system simulation. Appendix G provides all of the tables used in producing the summary graphs seen in the following subsection.

5.7.2 System Description

We see from Figure 60 that Chatoyant is the initial source of events since the piece-wise linear source stars are the only producers of the initial events. Therefore, we can predict that this system will behave much like the single and multi-bit Chatoyant to ModelSim test systems in terms of performance. We first prove this by examining the events per second measurement for the different simulations modes. Figure 62 summarizes the events per second measurements for the 1ps lock-step, the 1ns lock-step, and dynamic co-simulations as well as the non co-simulation results. For the standalone simulation, we base our comparison on a simulation performed without co-simulation in Chatoyant since it is the primary source of events.

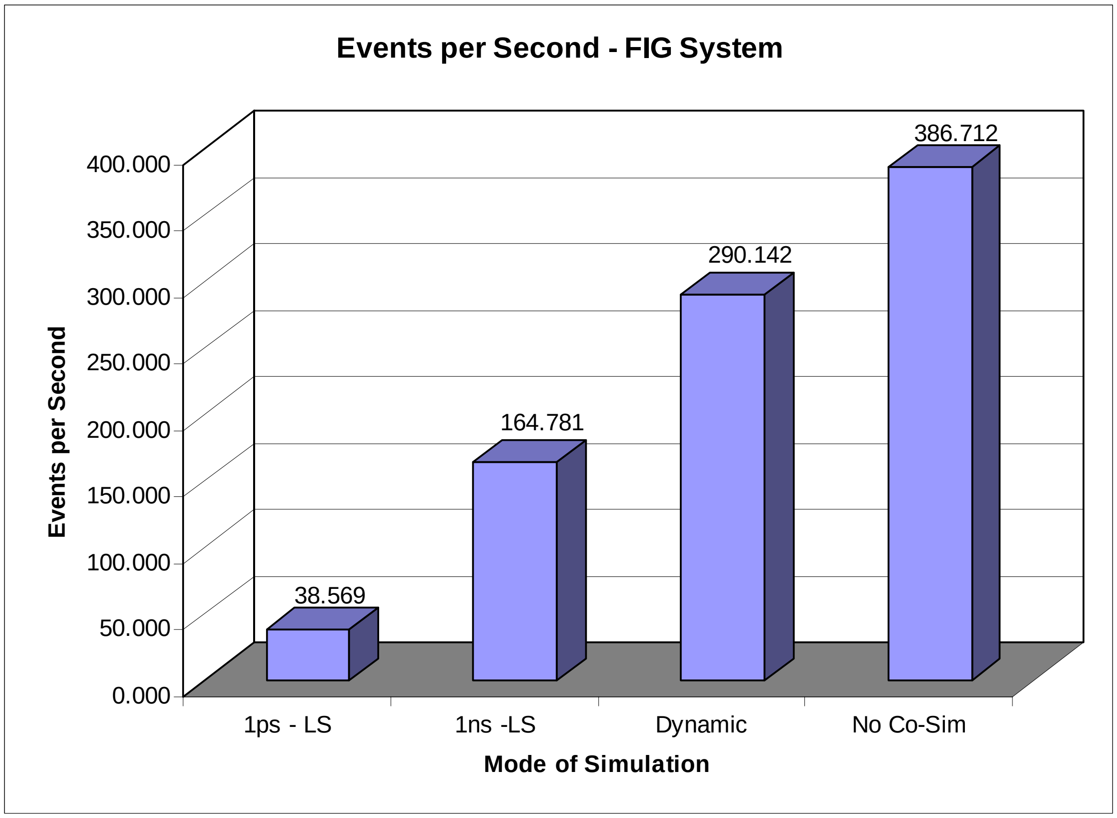


Figure 62 Events per second measurement for the FIG simulation

The results seen in Figure 62 show an interesting trend in terms of the performance of the co-simulation interface when compared to the non co-simulation execution. The non co-simulation events per second is 1.33X faster than the dynamic co-simulation. Likewise, the dynamic co-simulation mode is 1.76X faster than the 1ns lock-step co-simulation and 7.52X faster than the 1ps lock-step co-simulation. These numbers imply that the overhead of Chatoyant producing the events for this system and processing the ones coming in from the interface is greater than the overhead of the interface itself.

To support the assertion that the other Chatoyant processing is producing an overhead that is greater than the interface's overhead, we examine the code profile of the FIG simulation for the dynamic co-simulation mode, seen in Figure 63.

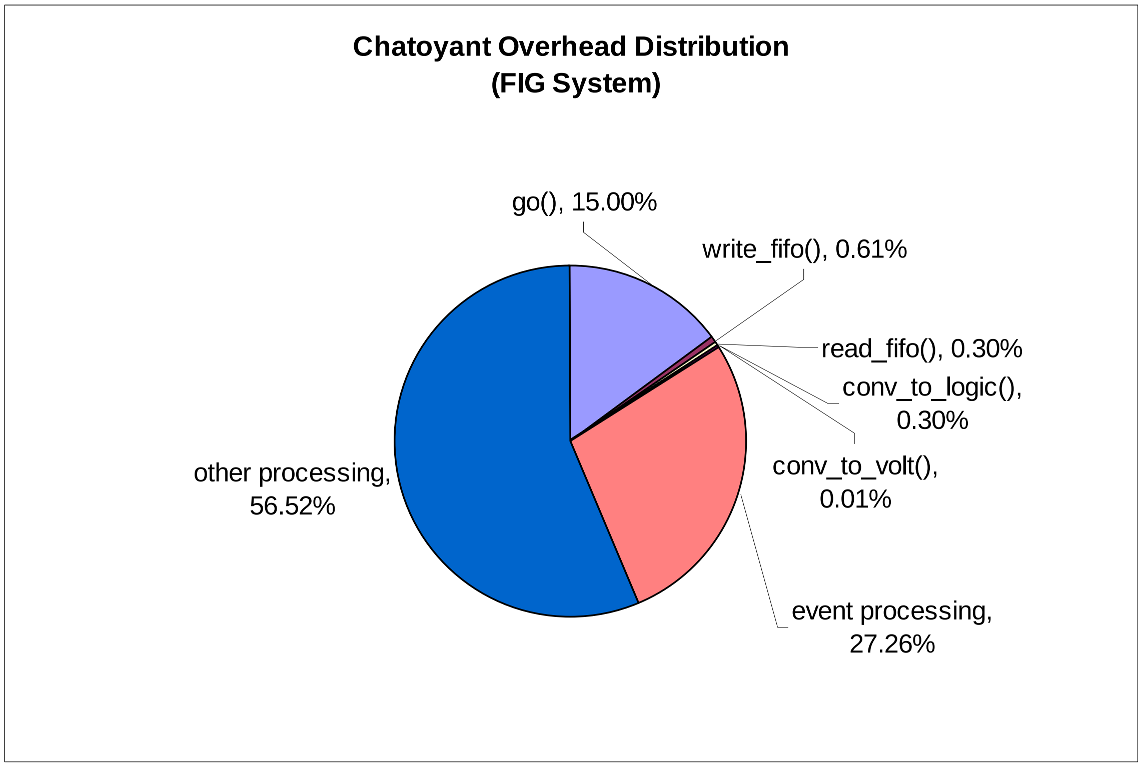


Figure 63 Chatoyant code profile for the FIG co-simulation in dynamic mode

As predicted, the additional Chatoyant processing is the greatest quantity of execution time at 56.52%. The event processing overhead produced by the interface and the piece-wise linear sources produce the second highest overhead at 27.26% of the total execution time. The interface code accounts for approximately 16.2% of the total execution time. This supports our previous assertion that the interface is not the primary consumption of execution time from the point of view of Chatoyant's runtime performance.

In terms of ModelSim, we see a majority of the execution time being devoted to the interface code. However, there is still are large percentage of execution spent on the other processing within the ModelSim kernel, which is most likely due to the number of gate level constructs. We see the overall breakdown of the code profile in Figure 64.

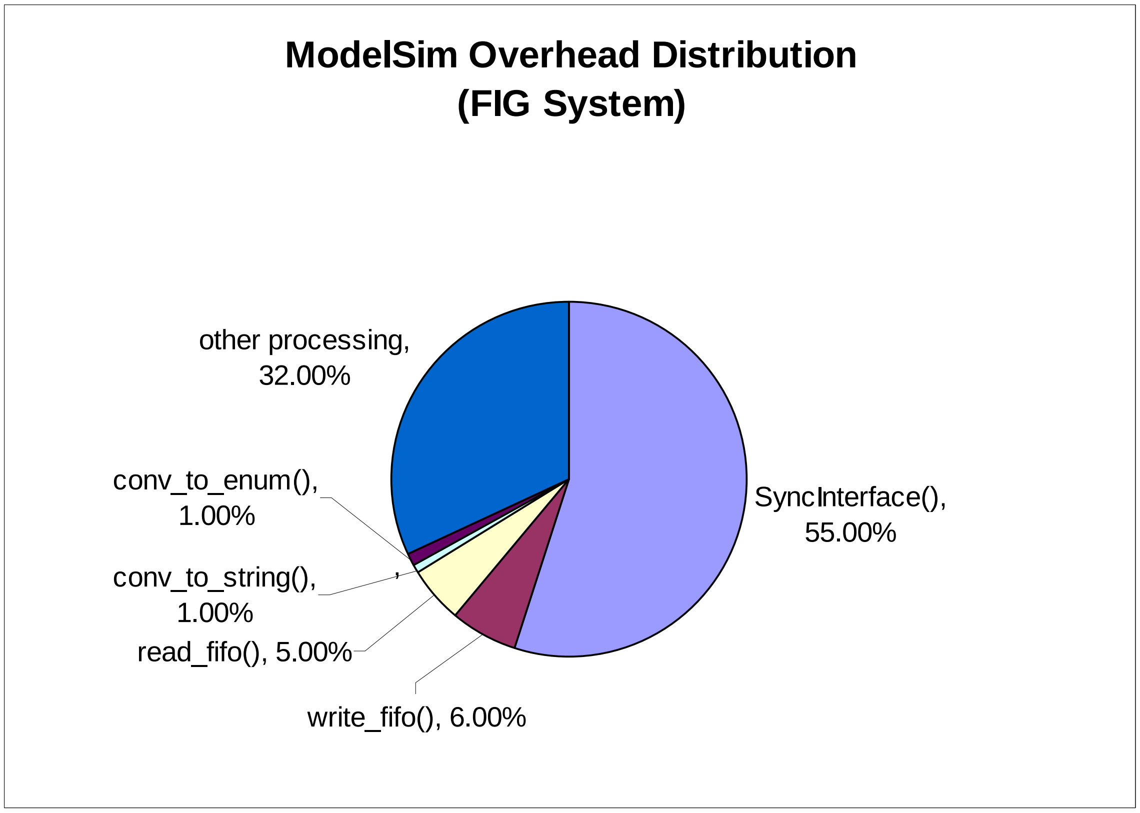


Figure 64 ModelSim code profile for the FIG co-simulation in dynamic mode

Our final analysis for this system is the waveforms produced in the dynamic co-simulation. The first waveform we examine is from the ModelSim side of the co-simulation, seen in Figure 65. Here we see the transitions of the input slices, which originated in Chatoyant, and their toggling at a future time. The clock in this system is used only for loading the control data and flagging the main controller that the configuration is complete. The snapshot in Figure 65 time represents a small period in time after the configuration is complete.

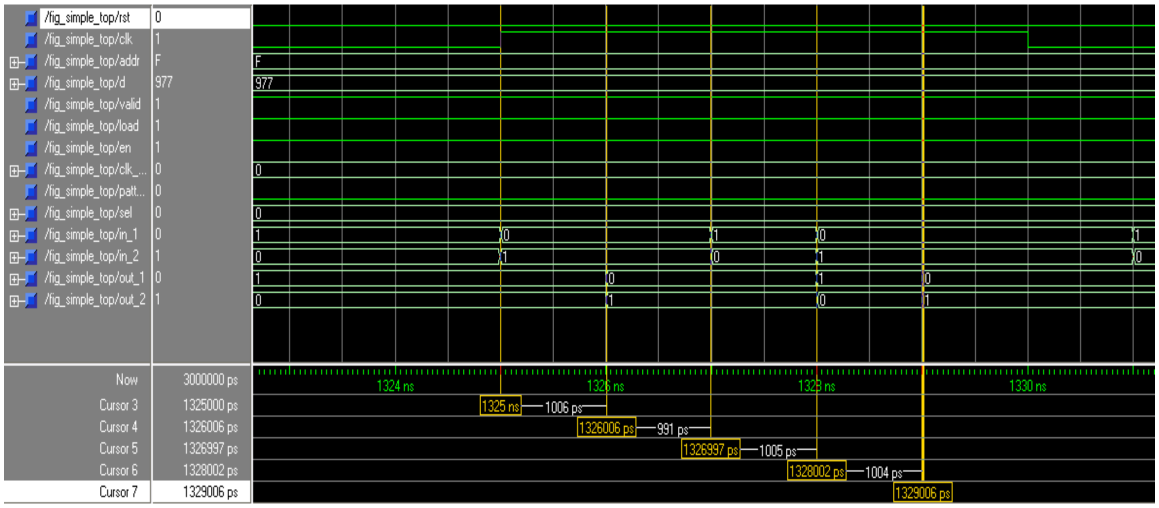


Figure 65 ModelSim waveform for the FIG co-simulation in dynamic mode

The Chatoyant waveform shows the incoming events as they arrive from the interface. The ModelSim bound events are seen in the in the previous figure as the in_1 and in_2 buses. Figure 66 shows the resulting Chatoyant waveforms.

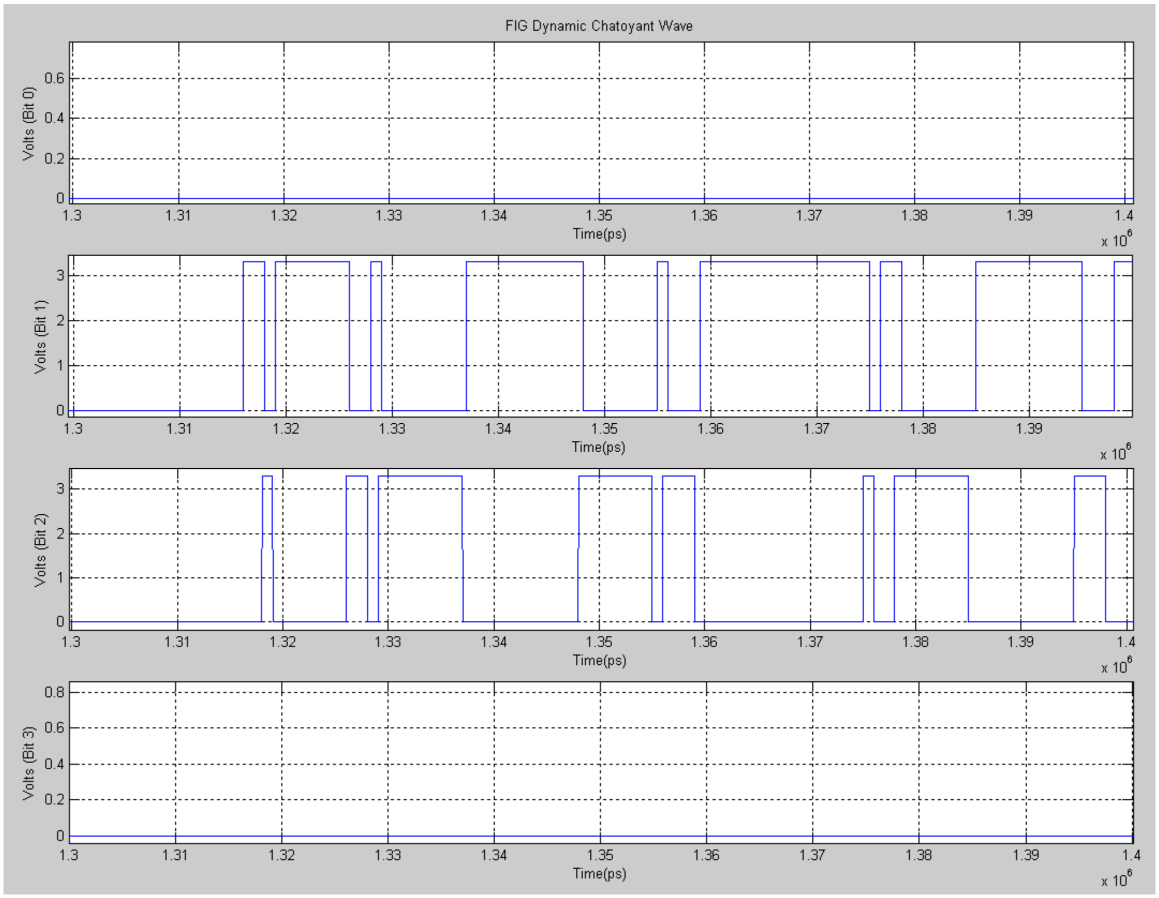


Figure 66 Chatoyant waveform for the FIG co-simulation using the dynamic mode

We now conclude the discussion on the FIG system. As reported, the co-simulation interface had an overall performance that was comparable to the standalone Chatoyant simulation. We determined that this is due to the overhead of producing and handling events within the components in the Chatoyant simulation itself. We also gained an insight into the accuracy of the simulation with the waveforms provided in Figures 65 and 66. We will now compare the results of this and previous sections to determine the scalability of the interface and how the performance compares overall for the dynamic mode of co-simulation.

5.8 ANALYSIS OF RESULTS

In this section we further analyze the data collected and examined in the previous sections by comparing the events per second and user cycles per second measurements for the dynamic co-simulation mode. We choose this particular mode because it provides the best overall runtime performance with a tolerable overhead compared to the lock-step method. Our goal is to determine how the interface will scale and perform for different systems.

The events per second measurements for the simulation of the six systems discussed in this chapter are summarized in Figure 67.

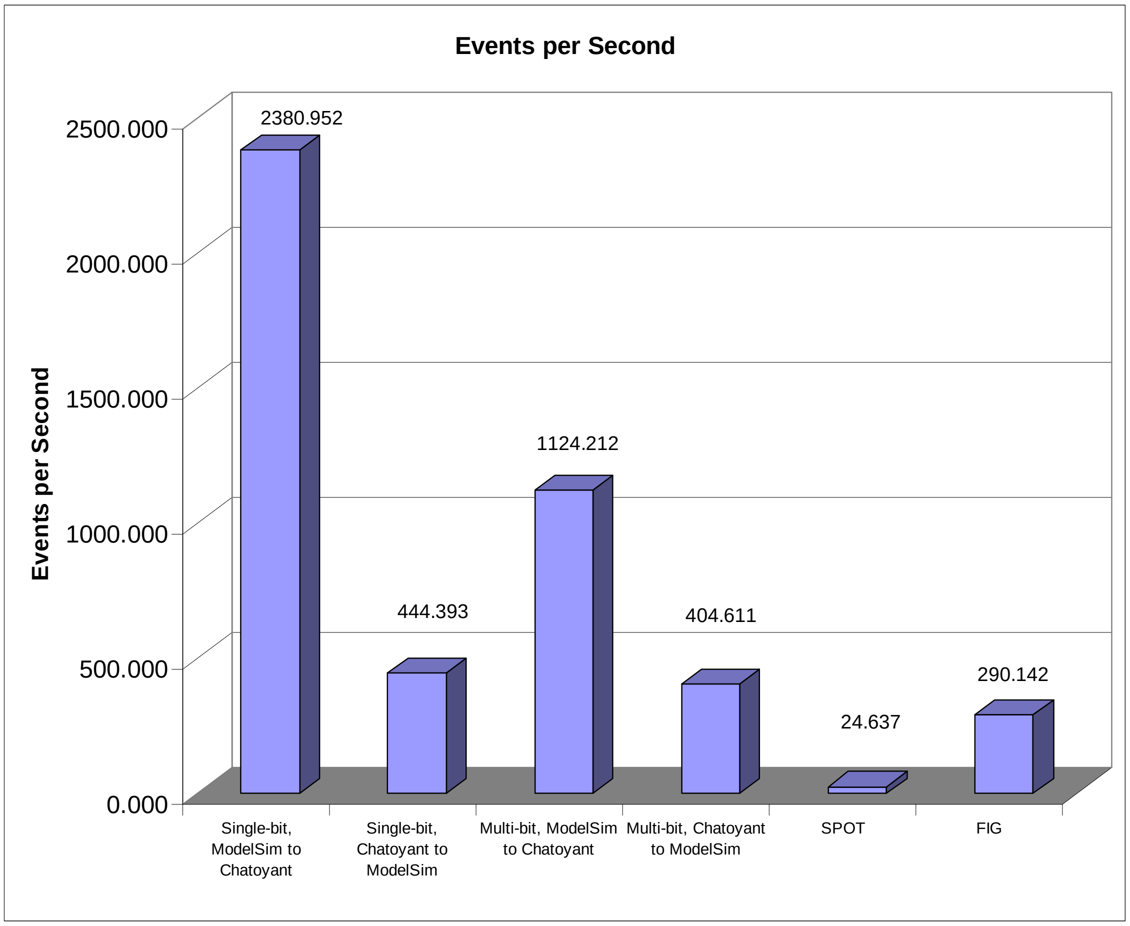


Figure 67 Summary of events per second for the six systems discussed in this chapter

From the graph in Figure 67 we can see that the highest performing systems for the co-simulation interface are the single and multi-bit ModelSim to Chatoyant tests. Since these systems are similar in terms of the behavior of the event traffic, we see that the overhead associated with handling buses is not that high when compared to the inclusion of an additional port. The multi-bit ModelSim to Chatoyant system differs from the single-bit version by the additional port in the interface. We can observe that the multi-bit bus performs at about one-half the level as the single-bit version.

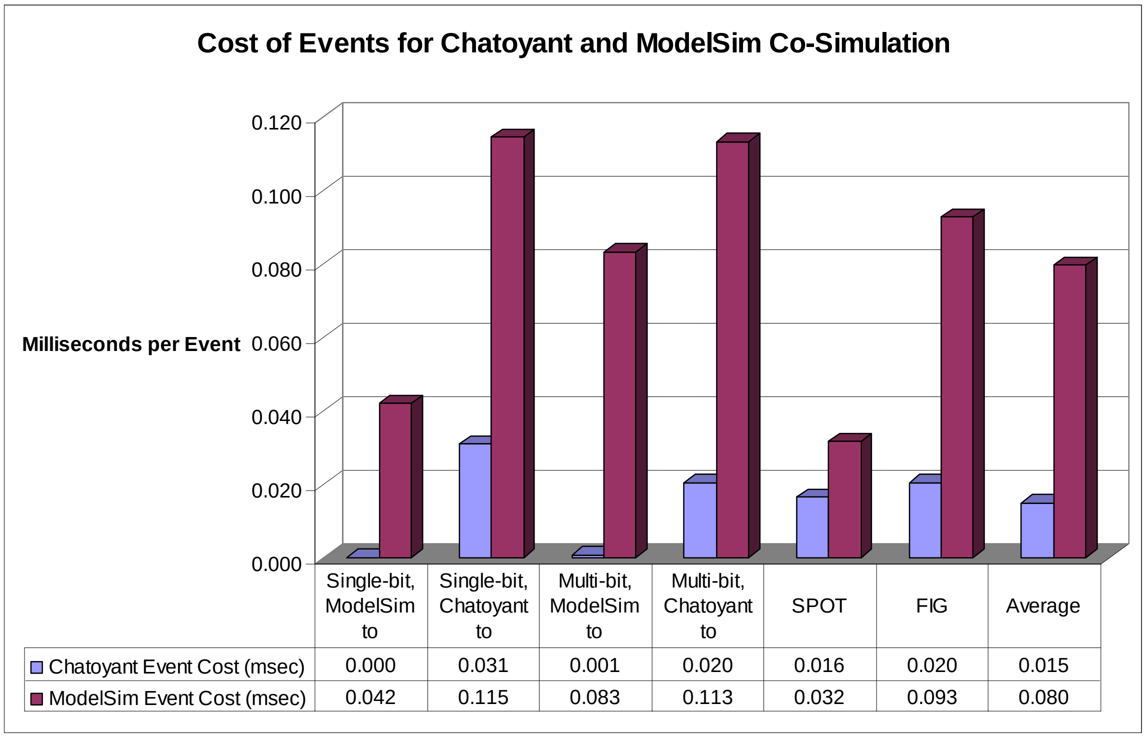
One issue that effects both the synchronization as well as Chatoyant event processing is the need to handle null events. These null events, within the context of the dynamic synchronization, occur for two reasons. First, for every event that goes from ModelSim to Chatoyant, there are two events created in the Chatoyant interface star. One event is the current state at the current time and the second event is the new state change at some user specified constant time later. This behavior is to create the rising or falling slope for an edge. Each of these events cause the Chatoyant star to re-evaluate the synchronization code found with in itself. From this we receive one null event based on the second event sent at the later time when the port's state is scheduled to change. The second source of null events is from the lookahead function returning an event that is not destined for any of the input ports of the interface star.

This second case for creating null events could lead to a potential optimization. If the DE scheduler's lookahead function was able to distinguish which events are destined for the interface star, we can filter all other events from the lookahead and potentially reduce the number of null events. Since the interface code is called when a event is present at one of its portholes, no caution is needed for interim events that occur from before the last determined synchronization. To implement this optimization, the DE scheduler needs to be aware of what ports are the inputs to the interface star. Then, every time a lookahead operation is performed, the lookahead method determines whether an event that has a time stamp greater than the current time is destined for an input port of the interface star. If it is, then the timestamp of the event is a valid lookahead time. Otherwise, we continue to search the event queue for a valid time. The only cost of this operation is the search through the event queue for finding a valid event and timestamp. More investigation is needed to determine the cost to benefit ratio of this new lookahead function versus the current overhead incurred from the null events.

When comparing FIG to SPOT, we see that FIG outperforms SPOT, even though there is more event traffic in FIG. Since ModelSim is the primary source in SPOT, the overhead of the interface is more pronounced than in FIG, where Chatoyant is the primary source of events. Another point to note is that SPOT shows an instance of both clock and data based events crossing the interface. The inclusion of a clock severely limits the achievable performance, especially high frequency clocks (100 to 1000 MHz). This is due to the high density of event traffic generated by the clock as well as the intermittent data event traffic. In FIG the event traffic was less dense and more random when compared to SPOT. This coupled with the higher computation load in the ModelSim simulator and event processing overhead found in Chatoyant provides an insight into how the interface will perform under the circumstance of a clock crossing the interface.

From this analysis and the data presented in the previous sections, we are able to determine the overall cost of sending one event over the interface. The cost of an event across the interface is determined as the impact it yields on the Chatoyant and ModelSim simulators. We must differentiate the impact of the cost analysis from the perspectives of Chatoyant and ModelSim since the interface only exists as segment of the total simulations in both simulators. This means that the interface is not a standalone application executing concurrently with Chatoyant and ModelSim.

To better quantify the cost in Chatoyant and ModelSim, we are able to calculate the time it takes to process one event in the interface. This value is the time spent in the interface code of both simulators divided by the events per second. This calculation is performed for both simulators since each has a different percentage of execution devoted to the interface. We show the number of milliseconds spent on each event for all six systems simulated in this chapter as well as an average cost in Figure 68.

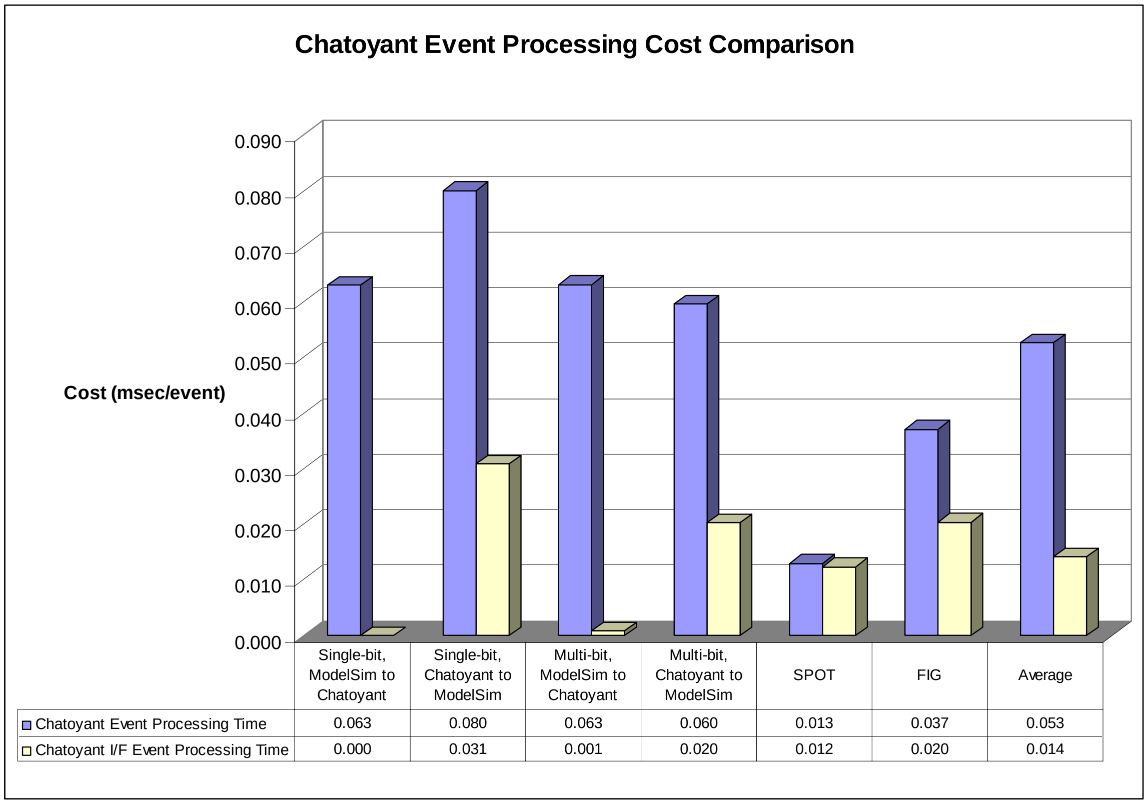


	Single-bit, ModelSim to	Single-bit, Chatoyant to	Multi-bit, ModelSim to	Multi-bit, Chatoyant to	SPOT	FIG	Average
▨ Chatoyant Event Cost (msec)	0.000	0.031	0.001	0.020	0.016	0.020	0.015
▨ ModelSim Event Cost (msec)	0.042	0.115	0.083	0.113	0.032	0.093	0.080

Figure 68 Time calculations spent on an event in the co-simulation interface for both Chatoyant and ModelSim

For all of the systems tested we see a low rate of variability in the time spent on processing an event in the interface. Some of the variability is due to the accuracy of the profiling functions performed on the code during runtime as well as environment variations from one simulation to the next. One constant trend seen in these six systems is that the cost of an event is higher in ModelSim than in Chatoyant. We can assert that an average runtime cost for an event is approximately five times greater in ModelSim than in Chatoyant, as seen in the average column in Figure 68.

As we have seen from the data presented in the six systems analyzed, the impact of an event is less on Chatoyant's performance than on ModelSim's performance. In all of the execution time distribution graphs, we see that the amount of time spent handling events in the interface does not account for the majority of the execution time. Event processing along with the other processing performed by other components of Chatoyant always accounts for highest percentage of the runtime for a simulation. We thus conclude that the cost of an event being processed within the interface star in Chatoyant is much smaller than the remainder of the simulation execution. We can further compare the event processing time within the interface with the overall Chatoyant event processing time. We show this in Figure 69.



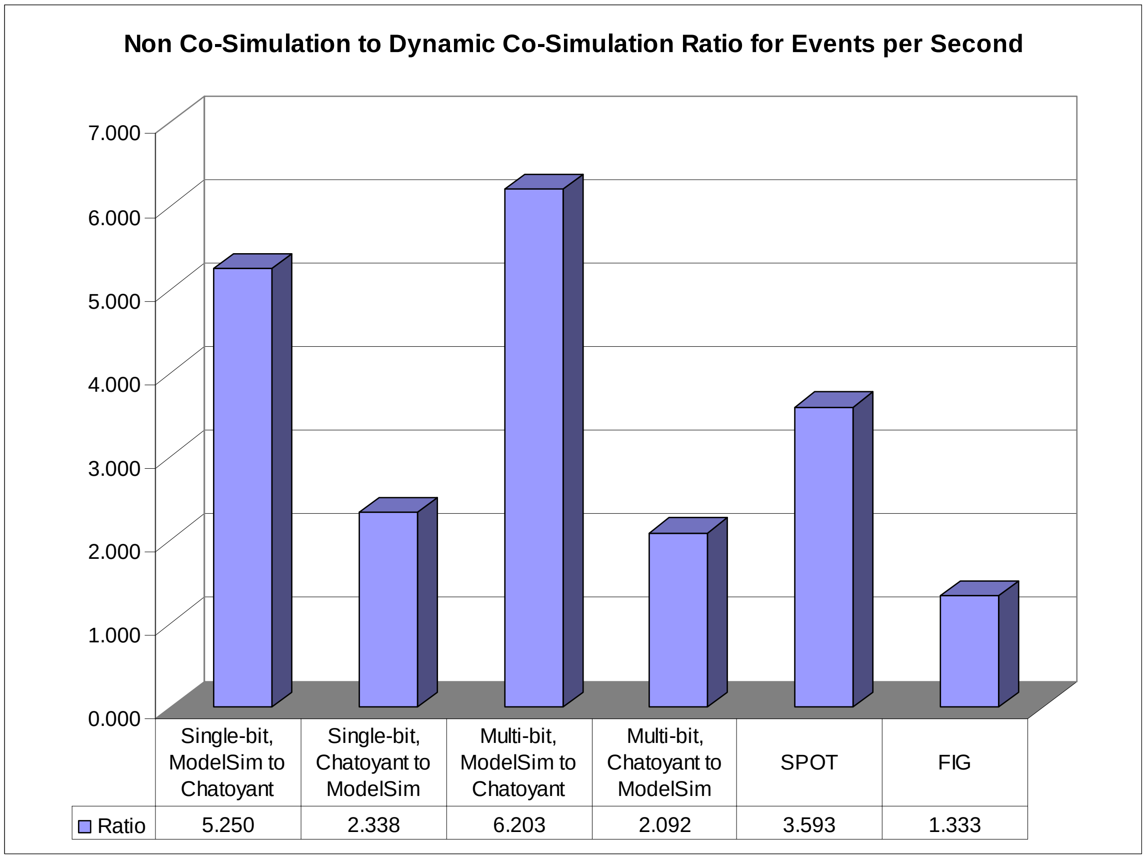
	Single-bit, ModelSim to Chatoyant	Single-bit, Chatoyant to ModelSim	Multi-bit, ModelSim to Chatoyant	Multi-bit, Chatoyant to ModelSim	SPOT	FIG	Average
🟦 Chatoyant Event Processing Time	0.063	0.080	0.063	0.060	0.013	0.037	0.053
🟨 Chatoyant I/F Event Processing Time	0.000	0.031	0.001	0.020	0.012	0.020	0.014

Figure 69 Event processing cost comparison between Chatoyant and the co-simulation interface within Chatoyant

The graph seen in Figure 69 indicates that the cost of processing an event in Chatoyant is higher than processing an event within the Chatoyant side of the co-simulation interface. One primary trend seen in this graph is that as the computational blocks perform more operations and/or become greater in number, the event processing time, in both Chatoyant and the co-simulation interface, decreases. An interesting anomaly is seen with the SPOT system when it is compared to the other systems. In the case of SPOT, the event processing time in both the interface and Chatoyant are almost equal, being within 1µs of each other. Initial examination indicates that this is most likely due to the configuration of SPOT having ModelSim as the initial source of events as well as the loop back within Chatoyant back into ModelSim. Therefore, the only events Chatoyant is required to process are the ones produced and consumed by the interface, which, in this particular case, are the same events. Therefore, the processing time for both Chatoyant and the interface are equal. Further examination is needed to better understand this behavior as well as how it relates to the other systems analyzed.

ModelSim, however, indicates the opposite cost per event relationship than what is seen in Chatoyant. For all of the simulations performed on the six systems analyzed, the overhead of processing events accounted for the highest percentage of the execution time during a simulation. This indicates that cost of processing an event in the interface is higher than any other operation within ModelSim from its perspective.

One final comparison we make is between the non co-simulation and the dynamic method events per second measurements. By taking the ratio between the two modes, we can see the slow down factor between using no co-simulation and the dynamic co-simulation. A lower the ratio implies a runtime of the dynamic co-simulation being closer to the non co-simulation. Figure 70 shows this ratio for all six systems.



	Single-bit, ModelSim to Chatoyant	Single-bit, Chatoyant to ModelSim	Multi-bit, ModelSim to Chatoyant	Multi-bit, Chatoyant to ModelSim	SPOT	FIG
▪ Ratio	5.250	2.338	6.203	2.092	3.593	1.333

Figure 70 Ratio of non co-simulation to dynamic co-simulation events per second

Figure 70 shows two major trends. First, the systems where Chatoyant is the primary source of events shows a closer ratio of events per second than the systems that have ModelSim as the primary source of events. This most likely due to the larger percentage of Chatoyant's time being spent on processing events and performing other operations, whereas ModelSim has more time devoted to the interface than any other process in itself. Secondly, when the system becomes more complex, such as in the instances of FIG and SPOT, the ratio approaches 1. This indicates that the co-simulation interface, in dynamic mode, has good scalability for larger systems, and thus does not impose a large decrease in system performance.

To conclude this chapter, we presented and analyzed the runtime performance of six systems. The four smaller systems provided measurements for both single and multi-bit systems with ModelSim and Chatoyant alternating as the source of events. This then lead into the analysis of two larger systems that combined the features of the first four systems. We examined the events per second, code profiles, and waveforms for each system. We finally performed a cross analysis of all six systems.

This cross analysis showed us that the cost of processing events in the interface star in Chatoyant is much smaller than the other processing being performed in Chatoyant. This analysis also showed us that the cost of processing events within the ModelSim interface code is much higher than the remaining processing time of ModelSim. From this we can conclude that as we simulate larger systems using Chatoyant and ModelSim, concurrently, making use of the dynamic co-simulation interface, Chatoyant is not impeded by the interface, however, ModelSim does see a performance limitation from the interface.

One final conclusion to be drawn from the analysis performed on all six systems is that utilizing the conservative asynchronous PDES technique, we are able to speedup co-simulation when compared to a lock-step method. By utilizing a lookahead method in each simulator, we are able to overcome potential deadlocks seen in conservative based asynchronous PDES, and show a speedup over the lock-step co-simulation, similar to the speedup seen between synchronous PDES and asynchronous PDES.

6.0 CONCLUSION AND FUTURE WORK

In this chapter we restate the purpose of this thesis, review the work performed, and summarize the results that were collected and how these results support that our goal has been met. We finish this chapter with a look into the future work of this research.

6.1 PURPOSE RESTATED

As first stated in the introduction, the purpose of this thesis was to develop a co-simulation interface utilizing inter-process communication mechanisms and techniques seen in Parallel Discrete Event Simulation. We implemented and tested this interface by using Chatoyant, a mixed signal, multi-domain simulator, built on the Ptolemy simulation framework, and ModelSim, a mixed language HDL simulator. We used these simulators since Ptolemy, and therefore Chatoyant, did not support direct VHDL or Verilog simulation within the Ptolemy framework. Since Chatoyant is a mixed-signal, multi-domain simulator, digital designs in an HDL format need to be supported.

6.2 THESIS SUMMARY

In summary, Chapter 2 presented the ModelSim mixed language HDL simulator as well as the Ptolemy multi-domain simulation framework and the Chatoyant mixed signal MOEMS simulator built on Ptolemy.

Chapter 3 first reviewed the theory behind discrete event simulation and parallel discrete event simulation methodologies. We then applied this foundation to the co-simulation interface between Chatoyant and ModelSim. We also examined how signals convert between the digital logic and analog signal realm during within the co-simulation interface. We concluded the chapter with a look into how the interface's performance is measured.

Chapter 4 explained the detail behind the architecture of the Chatoyant/ModelSim co-simulation interface as well as its implementation. We discussed the implementation of both the Chatoyant interface as well as the ModelSim FLI code for the lock-step and dynamic synchronization methods. We concluded this chapter by presenting a tool for automatically building the necessary files for the Chatoyant/ModelSim co-simulation interface from a user provided VHDL file.

Chapter 5 examined six systems that each provided a unique insight into how the interface performs. We analyzed the events per second, user cycles per second, the code profiles for both sides of the interface, and the waveforms within both simulators. We were able to draw conclusions, discussed later in Section 6.4, based on these results in the final section of the chapter. We now proceed with restating the work performed in this thesis.

6.3 WORK PERFORMED

We addressed the purpose of this thesis by applying the concepts found in Chapter 3 to the co-simulation interface architecture. We used both synchronous and asynchronous PDES theory to implement a lock-step (synchronous) and dynamic (asynchronous) mode of synchronization in the co-simulation interface. We then tested these different modes on six systems to determine which provided the best overall runtime performance. We see in the following section a summary of the conclusions we were able to draw from the results collected from the different simulations executed.

6.4 CONCLUSIONS FROM RESULTS

As discussed in the final section of Chapter 5, there are two major conclusions to be drawn. First, the impact of the co-simulation interface using the dynamic synchronization yielded a low cost per event within Chatoyant when compared to the other processing being performed within Chatoyant. Second, the cost per event for co-simulation was high in ModelSim when compared to the other processing being performed by ModelSim. These two conclusions provide insight as to how the interface scales for larger systems in both simulators. One final point is that we were able to provide speedup using the dynamic synchronization approach, which utilizes conservative asynchronous PDES, over a lock-step synchronization mechanism. This showed that were able to reduce the overhead of co-simulation by capitalizing on the techniques that provide the speedups in parallel and distributed discrete event simulation.

6.5 FUTURE WORK

There are two areas where an improvement can readily be detected. The first of these is in the lookahead method within Chatoyant and DE Scheduler. Currently, a lookahead operation will return the time stamp of the next event bin within the DE event queue. Event though this does provide a speedup over the lock-step methods, There is still unnecessary overhead induced by the interface star rescheduling itself at timestamps of future events that do not directly impact the interface and the temporal causality between the two simulators. Therefore, a more efficient approach could be taken to resolve this issue. The DE scheduler, upon initialization, can provide a list of all signals that directly effect the interface. Then, during a simulation, when a lookahead is performed, the scheduler only needs to determine when the next event time for an event destined to the interface star is going to occur, and thus only require the interface star to re-execute its *go()* method at the next time an event will occur on one of the ports of the star.

Another improvement requires a more drastic change. Instead of performing event based co-simulation, a transaction based approach can be used. This requires less communication and synchronization between the simulators while providing a higher level of abstraction when considering how the different ends of system operate. For example, in the case of a larger system where a standardized bus such as PCI is used, the transactions can reduce the quantity of data packets and synchronizations produced within a given window of time, defined as one of the parameters of specific transaction. This could potentially speedup the co-simulation to an even greater degree and provides an architectural methodology for exploring the design in question.

APPENDIX A - TERMS

- Chatoyant – A mixed signal, multi-domain simulator developed at the University of Pittsburgh
- DE – Discrete Event
- DES – Discrete Event Simulation
- DDF – Dynamic Data-Flow, a domain and type of simulation based on the order and flow of data. (9)
- Events per Second – The number of discrete events that are registered in one second of execution.
- HDL – Hardware Description Language
- IPC – Inter-Process Communication
- LP – Logical Process, in the scope of simulation, is the part of a simulator that performs some computation on event data to generate more event data.
- MCM – Multi-Chip Module
- MNA – Modified Nodal Analysis
- MEMS – Micro-Electro-Mechanical Systems
- ModelSim – A mixed HDL simulator from Mentor Graphics
- MOEMS – Micro-Opto-Electro-Mechanical Systems
- PDES – Parallel Discrete Event Simulation

- Ptolemy – A multi-domain simulation framework from the University of California, Berkeley (9)
- Scheduler – The part of a simulator that controls event ordering and distribution
- SDF – Synchronous Data Flow, similar to DDF except with a predetermined static schedule (9)

APPENDIX B – SINGLE-BIT MODELSIM TO CHATOYANT RESULTS

Table 2 Single-bit, ModelSim to Chatoyant Lock-Step Data

Lock-Step:

Sync	Wall-Clock	Chatoyant - Received			Chatoyant - Transmitted		
Resolution	Runtime	*Events*	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
1ps	38.1519	400	997600	998000	0	998001	998001
1ns	2.348	286	999	1285	0	999	999
Sync	▨	**ModelSim -Received**			**ModelSim - Transmitted**		
Resolution	▨	*Events*	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
1ps	▨	0	998001	998001	400	997600	998000
1ns	▨	0	999	999	286	999	1285

Table 3 Single-bit, ModelSim to Chatoyant Lock-Step events per second

Sync Resolution	Events per *Second*
1ps - LS	10.484
1ns - LS	121.806

Table 4 Single-bit, ModelSim to Chatoyant 1ps lock-step code profile

1ps:

Chatoyant	% of Time	ModelSim	% of Time
go()	36.52%	*SyncInterface()*	33.00%
write_fifo()	2.77%	*write_fifo()*	13.00%
read_fifo()	1.24%	*read_fifo()*	12.00%
conv_to_volt()	1.00%	*conv_to_string()*	0.00%
conv_to_logic()	0.00%	*conv_to_enum()*	0.00%
other	58.47%	*other*	42.00%
Total	*100.00%*	*Total*	*100.00%*

Table 5 Single-bit, ModelSim to Chatoyant 1ns lock-step code profile

1ns:

Chatoyant	% of Time	ModelSim	% of Time
go()	50.00%	*SyncInterface()*	23.00%
write_fifo()	1.00%	*write_fifo()*	11.00%
read_fifo()	1.00%	*read_fifo()*	9.00%
conv_to_volt()	1.00%	*conv_to_string()*	0.00%
conv_to_logic()	0.00%	*conv_to_enum()*	0.00%
other	47.00%	*other*	57.00%
Total	*100.00%*	*Total*	*100.00%*

Table 6 Single-bit, ModelSim to Chatoyant dynamic co-simulation results

Dynamic

Wall-Clock Runtime	**Chatoyant - Received** *Events*	*Nulls*	*Total*	**Chatoyant - Transmitted** *Events*	*Nulls*	*Total*
0.168	400	1200	1600	0	1600	1600
	ModelSim - Received *Events*	*Nulls*	*Total*	**ModelSim - Transmitted** *Events*	*Nulls*	*Total*
	0	1600	1600	400	1200	1600

Table 7 Single-bit, ModelSim to Chatoyant dynamic co-simulation events per second

Events per *Second*
2380.952

Table 8 Single-bit, ModelSim to Chatoyant dynamic co-simulation code profile

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	0.01%	*SyncInterface()*	50.00%
write_fifo()	0.01%	*write_fifo()*	0.00%
read_fifo()	0.01%	*read_fifo()*	0.00%
conv_to_volt()	0.01%	*conv_to_string()*	0.00%
conv_to_logic()	0.01%	*conv_to_enum()*	0.00%
event processing	75.00%		
other processing	24.95%	*other processing*	50.00%
Total	*100.00%*	*Total*	*100.00%*

Table 9 Single-bit, ModelSim to Chatoyant event processing profile for Chatoyant running the dynamic co-simulation

Chatoyant Runtime Overhead - Event Processing Functions:

% of Overhead	*Function Name*
25	SimControl::flagValues()
25	KnownListEntry::~KnownListEntry [in-charge]()
25	NamedObj::name() const
75	

Table 10 Single-bit, ModelSim to Chatoyant runtime and events per second for no co-simulation in ModelSim

No Co-Sim

	ModelSim
Runtime	0.032
Events per Sec	12500.000

APPENDIX C – SINGLE-BIT CHATOYANT TO MODELSIM RESULTS

Table 11 Single-bit, Chatoyant to ModelSim Lock-Step Data

Lock-Step:

Sync Resolution	Wall-Clock Runtime	Chatoyant - Received			Chatoyant - Transmitted		
		Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
1ps	37.3291	0	1000001	1000001	400	999200	999600
1ns	15.264	0	1001	1001	400	9200	9600
Sync Resolution		**ModelSim -Received**			**ModelSim - Transmitted**		
		Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
1ps		400	999200	999600	0	1000001	1000001
1ns		400	9200	9600	0	1001	1001

Table 12 Single-bit, Chatoyant to ModelSim Lock-Step events per second

Sync Resolution	Events per *Second*
1ps - LS	10.716
1ns - LS	26.205

Table 13 Single-bit, Chatoyant to ModelSim 1ps lock-step code profile

1ps:

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	33.77%	*SyncInterface()*	31.00%
write_fifo()	3.08%	*write_fifo()*	16.00%
read_fifo()	0.81%	*read_fifo()*	15.00%
conv_to_volt()	0.00%	*conv_to_string()*	0.00%
conv_to_logic()	0.16%	*conv_to_enum()*	0.00%
other	62.18%	*all else*	38.00%
Total	*100.00%*	*Total*	*100.00%*

Table 14 Single-bit, Chatoyant to ModelSim 1ns lock-step code profile

1ns:

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	26.00%	*SyncInterface()*	27.00%
write_fifo()	1.02%	*write_fifo()*	13.00%
read_fifo()	0.44%	*read_fifo()*	11.00%
conv_to_volt()	0.00%	*conv_to_string()*	0.00%
conv_to_logic()	0.16%	*conv_to_enum()*	0.00%
other	72.38%	*other*	49.00%
Total	*100.00%*	*Total*	*100.00%*

Table 15 Single-bit, Chatoyant to ModelSim dynamic co-simulation results

Dynamic

Wall-Clock Runtime	Chatoyant - Received			Chatoyant - Transmitted		
	Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
0.899	0	1997	1997	400	1598	1998
	ModelSim - Received			**ModelSim - Transmitted**		
	Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
	400	1598	1998	0	1997	1997

Table 16 Single-bit, Chatoyant to ModelSim dynamic co-simulation events per second

Events per *Second*
444.939

Table 17 Single-bit, Chatoyant to ModelSim dynamic co-simulation code profile

Chatoyant	% of Time	ModelSim	% of Time
go()	13.79%	*SyncInterface()*	58.00%
write_fifo()	3.45%	*write_fifo()*	4.00%
read_fifo()	3.45%	*read_fifo()*	4.00%
conv_to_volt()	0.00%	*conv_to_string()*	0.00%
conv_to_logic()	0.00%	*conv_to_enum()*	4.00%
event processing	35.65%		
other processing	43.66%	*other processing*	30.00%
Total	*100.00%*	*Total*	*100.00%*

Table 18 Single-bit, Chatoyant to ModelSim event processing profile for Chatoyant running the dynamic co-simulation

Chatoyant Runtime Overhead - Event Processing Functions:

% of Overhead	*Function Name*
6.9	FloatParticle::operator double() const
1.72	ListIter::next()
4.62	SimControl::flagValues()
0	SimControl::doPostActions(Star*)
3.45	Queue::getHead()
3.45	SimControl::haltStatus()
3.45	SimControl::getPollFlag()
0	SimControl::doPreActions(Star*)
3.45	SimControl::haltRequested()
0	FloatParticle::die()
0	FloatParticle::operator=(Particle const&)
3.45	Queue::head()
1.72	ListIter::ListIter[in-charge](SequentialList const&)
1.72	PortHole::receiveData()
1.72	PortHole::sendData()
0	GenericPort::isItOutput() const
0	GenericPort::isItInput() const
35.65	

Table 19 Single-bit, Chatoyant to ModelSim runtime and events per second for no co-simulation in Chatoyant

No Co-Sim

	Chatoyant
Runtime	0.385
Events per Sec	1038.961

APPENDIX D – MULTI-BIT MODELSIM TO CHATOYANT RESULTS

Table 20 Multi-bit, ModelSim to Chatoyant lock-step data

Lock-Step:

Sync	**Wall-Clock**	**Chatoyant - Received**			**Chatoyant - Transmitted**		
Resolution	**Runtime**	*Events*	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
1ps	41.9313	252	999800	1000052	0	1000001	1000001
1ns	5.37992	252	799	1051	0	1001	1001
Sync		**ModelSim -Received**			**ModelSim - Transmitted**		
Resolution		*Events*	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
1ps		0	1000001	1000001	252	999800	1000052
1ns		0	1001	1001	252	800	1052

Table 21 Multi-bit, ModelSim to Chatoyant lock-step events per second

Sync Resolution	**Events per** *Second*

▨	
1ps - LS	6.010
1ns - LS	46.841

Table 22 Multi-bit, ModelSim to Chatoyant 1ps lock-step code profile

1ps:

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	66.66%	*SyncInterface()*	68.00%
write_fifo()	0.00%	*write_fifo()*	7.00%
read_fifo()	0.00%	*read_fifo()*	9.00%
conv_to_volt()	0.00%	*conv_to_string()*	2.00%
conv_to_logic()	0.00%	*conv_to_enum()*	2.00%
other	33.34%	*other*	12.00%
Total	*100.00%*	*Total*	*100.00%*

Table 23 Multi-bit, ModelSim to Chatoyant 1ns lock-step code profile

1ns:

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	66.66%	*SyncInterface()*	43.00%
write_fifo()	0.00%	*write_fifo()*	10.00%
read_fifo()	0.00%	*read_fifo()*	8.00%
conv_to_volt()	0.00%	*conv_to_string()*	3.00%
conv_to_logic()	0.00%	*conv_to_enum()*	1.00%
other	33.34%	*other*	35.00%
Total	*100.00%*	*Total*	*100.00%*

Table 24 Multi-bit, ModelSim to Chatoyant dynamic co-simulation results

Dynamic

Wall-Clock Runtime	**Chatoyant - Received** *Events*	*Nulls*	*Total*	**Chatoyant - Transmitted** *Events*	*Nulls*	*Total*
0.337125	379	1601	1980	0	1802	1802
	ModelSim - Received *Events*	*Nulls*	*Total*	**ModelSim - Transmitted** *Events*	*Nulls*	*Total*
	0	1802	1802	379	1601	1980

Table 25 Multi-bit, ModelSim to Chatoyant dynamic co-simulation events per second

Events per *Second*
1124.212

Table 26 Multi-bit, ModelSim to Chatoyant dynamic co-simulation code profile

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	1.00%	*SyncInterface()*	99.00%
write_fifo()	0.00%	*write_fifo()*	0.00%
read_fifo()	0.00%	*read_fifo()*	0.00%
conv_to_volt()	0.00%	*conv_to_string()*	0.00%
conv_to_logic()	0.00%	*conv_to_enum()*	0.00%
event processing	75.00%		
other processing	24.00%	*other processing*	1.00%
Total	*100.00%*	*Total*	*100.00%*

Table 27 Multi-bit, ModelSim to Chatoyant event processing profile for Chatoyant running the dynamic co-simulation

Chatoyant Runtime Overhead - Event Processing Functions:

% of Overhead	*Function Name*
50	KnownBlock::findEntry(char const*, KnownListEntry*)
12.5	GenericPort::isItOutput() const
12.5	GenericPort::isItInput() const
75	

Table 28 Multi-bit, ModelSim to Chatoyant runtime and events per second for no co-simulation in ModelSim

No Co-Sim

	ModelSim
Runtime	0.059
Events per Sec	6372.961

APPENDIX E – MULTI-BIT CHATOYANT TO MODELSIM RESULTS

Table 29 Multi-bit, Chatoyant to ModelSim lock-step data

Lock-Step:

Sync Resolution	**Wall-Clock Runtime**	**Chatoyant - Received**			**Chatoyant - Transmitted**		
		Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
1ps	15.427	0	100001	100001	322	99840	100162
1ns	9.126	0	18440	18440	161	18280	18441
Sync Resolution		**ModelSim -Received**			**ModelSim - Transmitted**		
		Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
1ps		322	99840	100162	0	100001	100001
1ns		161	18280	18441	0	18280	18280

Table 30 Multi-bit, Chatoyant to ModelSim lock-step events per second

Sync Resolution	**Events per** *Second*
1ps - LS	20.872
1ns - LS	17.642

Table 31 Multi-bit, Chatoyant to ModelSim 1ps lock-step code profile

1ps:

Chatoyant	% of Time	ModelSim	% of Time
go()	28.79%	*SyncInterface()*	68.00%
write_fifo()	1.52%	*write_fifo()*	7.00%
read_fifo()	0.00%	*read_fifo()*	9.00%
conv_to_volt()	0.00%	*conv_to_string()*	1.00%
conv_to_logic()	0.00%	*conv_to_enum()*	2.00%
other	69.69%	*other*	13.00%
Total	*100.00%*	*Total*	*100.00%*

Table 32 Multi-bit, Chatoyant to ModelSim 1ns lock-step code profile

1ns:

Chatoyant	% of Time	ModelSim	% of Time
go()	4.17%	*SyncInterface()*	44.00%
write_fifo()	4.17%	*write_fifo()*	9.00%
read_fifo()	0.00%	*read_fifo()*	11.00%
conv_to_volt()	0.00%	*conv_to_string()*	1.00%
conv_to_logic()	0.00%	*conv_to_enum()*	2.00%
other	91.66%	*other*	33.00%
Total	*100.00%*	*Total*	*100.00%*

Table 33 Multi-bit, Chatoyant to ModelSim dynamic co-simulation results

Dynamic

Wall-Clock Runtime	**Chatoyant Received**			**Chatoyant - Transmitted**		
	Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
7.894	0	40910	40910	3194	39314	42508
	ModelSim -Received			**ModelSim - Transmitted**		
	Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
	3194	39314	42508	0	40910	40910

Table 34 Multi-bit, Chatoyant to ModelSim dynamic co-simulation events per second

Events per *Second*
404.611

Table 35 Multi-bit, Chatoyant to ModelSim dynamic co-simulation code profile

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	10.98%	*SyncInterface()*	43.00%
write_fifo()	0.29%	*write_fifo()*	8.00%
read_fifo()	0.29%	*read_fifo()*	10.00%
conv_to_volt()	0.00%	*conv_to_string()*	0.00%
conv_to_logic()	0.58%	*conv_to_enum()*	0.00%
event processing	32.11%		
other processing	55.75%	*other processing*	39.00%
Total	*100.00%*	*Total*	*100.00%*

Table 36 Multi-bit, Chatoyant to ModelSim event processing profile for Chatoyant running the dynamic co-simulation

Chatoyant Runtime Overhead - Event Processing Functions:

% of Overhead	*Function Name*
6.36	ListIter::next()
4.05	Star::asStar()
3.76	FloatParticle::operator double() const
2.31	SimControl::haltRequested()
2.02	FloatParticle::die()
1.73	SimControl::flagValues()
1.73	SimControl::haltStatus()
1.45	SimControl::getPollFlag()
1.45	FloatParticle::operator=(Particle const&)
1.16	SimControl::doPreActions(Star*)
1.16	Plasma::put(Particle*)
0.87	SimControl::setPollFlag()
0.87	Queue::putTail(void*)
0.58	badType(NamedObj&, Envelope&, char const*)
0.29	FloatParticle::initialize()
0.29	FloatParticle::operator<<(double)
0.29	FloatParticle::operator<<(int)
0.29	CriticalSection::~CriticalSection [in-charge]()
0.29	Queue::putHead(void*)
0.29	ListIter::ListIter[in-charge](SequentialList const&)
0.29	ListIter::operator++(int)
0.29	GenericPort::isItOutput() const
0.29	GenericPort::isItInput() const
32.11	

Table 37 Multi-bit, Chatoyant to ModelSim runtime and events per second for no co-simulation in Chatoyant

No Co-Sim

	Chatoyant
Runtime	3.774
Events per Sec	846.317

APPENDIX F - SMART PIXEL OPTICAL TRANSCEIVER (SPOT) RESULTS

Table 38 SPOT lock-step data

Lock-Step:

Sync Resolution	Wall-Clock Runtime	Chatoyant - Received Events	Nulls	Total	Chatoyant - Transmitted Events	Nulls	Total
1ps	1,605.67	96	174899	174995	288	171502	171790
1ns	842.65	96	125	221	288	1	289
Sync Resolution	▨	**ModelSim -Received** *Events*	*Nulls*	*Total*	**ModelSim - Transmitted** *Events*	*Nulls*	*Total*
1ps	▨	288	171502	171790	96	174899	174995
1ns	▨	288	1	289	96	125	221

Table 39 SPOT lock-step events per second

Sync Resolution	Events per *Second*
1ps - LS	0.239
1ns - LS	0.456

Table 40 SPOT 1ps lock-step code profile

1ps:

Chatoyant	% of Time	ModelSim	% of Time
go()	31.50%	*SyncInterface()*	95.00%
write_fifo()	3.10%	*write_fifo()*	1.25%
read_fifo()	3.55%	*read_fifo()*	1.25%
conv_to_volt()	0.50%	*conv_to_string()*	1.00%
conv_to_logic()	0.45%	*conv_to_enum()*	0.00%
other	60.90%	*other*	1.50%
Total	*100.00%*	*Total*	*100.00%*

Table 41 SPOT 1ns lock-step code profile

1ns:

Chatoyant	% of Time	ModelSim	% of Time
go()	22.46%	*SyncInterface()*	92.00%
write_fifo()	1.20%	*write_fifo()*	1.00%
read_fifo()	1.20%	*read_fifo()*	1.00%
conv_to_volt()	1.00%	*conv_to_string()*	1.00%
conv_to_logic()	1.00%	*conv_to_enum()*	0.00%
other	73.14%	*other*	5.00%
Total	*100.00%*	*Total*	*100.00%*

Table 42 SPOT dynamic co-simulation results

Dynamic

Wall-Clock	Chatoyant - Received			Chatoyant - Transmitted		
Runtime	*Events*	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
7.712	96	124306	124402	94	124309	124403
	ModelSim -Received			**ModelSim - Transmitted**		
	Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
	94	124309	124403	96	124306	124402

Table 43 SPOT dynamic co-simulation events per second

Events per *Second*
24.637

Table 44 SPOT dynamic co-simulation code profile

Chatoyant	% of Time	ModelSim	% of Time
go()	40.10%	*SyncInterface()*	59.00%
write_fifo()	1.04%	*write_fifo()*	10.00%
read_fifo()	4.17%	*read_fifo()*	9.00%
conv_to_volt()	0.00%	*conv_to_string()*	2.00%
conv_to_logic()	0.00%	*conv_to_enum()*	2.00%
event processing	43.72%		
other processing	10.97%	*other processing*	18.00%
Total	*100.00%*	*Total*	*100.00%*

Table 45 SPOT vent processing profile for Chatoyant running the dynamic co-simulation

Chatoyant Runtime Overhead -
Event Processing Functions:

% of Overhead	*Function Name*
22.92	FloatParticle::operato
6.25	ListIter::next()
1.04	SimControl::flagValues
1.04	FloatParticle::operato
0.52	ListIter::ListIter[in-
2.07	Queue::getHead()
1.04	Queue::putTail(void*)
0.52	PortHole::receiveData(
0.52	PortHole::sendData()
3.12	SimControl::haltStatus
2.08	SimControl::getPollFla
1.04	SimControl::haltReques
0.52	Queue::putHead(void*)
1.04	Plasma::put(Particle*)
43.72	

Table 46 SPOT runtime and events per second for no co-simulation in ModelSim

No Co-Sim

	ModelSim
Runtime	2.146
Events per Sec	88.524

APPENDIX G – FIBER IMAGE GUIDE (FIG) RESULTS

Table 47 FIG lock-step data

Lock-Step:

Sync Resolution	**Wall-Clock Runtime**	**Chatoyant - Received** *Events*	*Nulls*	*Total*	**Chatoyant - Transmitted** *Events*	*Nulls*	*Total*
1ps	130.674	677	2999661	3000338	4363	2995638	3000001
1ns	30.586	677	2961	3638	4363	2938	7301
Sync Resolution	▨	**ModelSim -Received** *Events*	*Nulls*	*Total*	**ModelSim - Transmitted** *Events*	*Nulls*	*Total*
1ps	▨	4363	2995638	3000001	677	2999661	3000338
1ns	▨	4363	2938	7301	677	2961	3638

Table 48 FIG lock-step events per second

Sync Resolution	**Events per** *Second*
1ps - LS	38.569
1ns -LS	164.781

Table 49 FIG 1ps lock-step code profile

1ps:

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	31.20%	*SyncInterface()*	89.00%
write_fifo()	3.21%	*write_fifo()*	1.00%
read_fifo()	3.80%	*read_fifo()*	1.00%
conv_to_volt()	0.11%	*conv_to_string()*	1.00%
conv_to_logic()	0.23%	*conv_to_enum()*	0.00%
other	61.45%	*other*	8.00%
Total	*100.00%*	*Total*	*100.00%*

Table 50 FIG 1ns lock-step code profile

1ns:

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	13.86%	*SyncInterface()*	76.00%
write_fifo()	1.00%	*write_fifo()*	1.00%
read_fifo()	1.00%	*read_fifo()*	1.00%
conv_to_volt()	1.00%	*conv_to_string()*	1.00%
conv_to_logic()	1.00%	*conv_to_enum()*	0.00%
other	82.14%	*other*	21.00%
Total	*100.00%*	*Total*	*100.00%*

Table 51 FIG dynamic co-simulation results

Dynamic

Wall-Clock Runtime	**Chatoyant - Received**			**Chatoyant - Transmitted**		
	Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
17.357	677	126523	127200	4359	122796	127155
	ModelSim -Received			**ModelSim - Transmitted**		
	Events	*Nulls*	*Total*	*Events*	*Nulls*	*Total*
	4359	122796	127155	677	126523	127200

Table 52 FIG dynamic co-simulation events per second

Events per *Second*
290.142

Table 53 FIG dynamic co-simulation code profile

Chatoyant	**% of Time**	**ModelSim**	**% of Time**
go()	15.00%	*SyncInterface()*	55.00%
write_fifo()	0.61%	*write_fifo()*	6.00%
read_fifo()	0.30%	*read_fifo()*	5.00%
conv_to_volt()	0.01%	*conv_to_string()*	1.00%
conv_to_logic()	0.30%	*conv_to_enum()*	1.00%
event processing	27.26%		
other processing	56.52%	*other processing*	32.00%
Total	*100.00%*	*Total*	*100.00%*

Table 54 FIG event processing profile for Chatoyant running the dynamic co-simulation

Chatoyant Runtime Overhead - Event Processing Functions:

% of Overhead	*Function Name*
6.06	ListIter::next()
4.24	FloatParticle::operator double() const
2.12	SimControl::getPollFlag()e)
1.21	_GLOBAL__D__ZN8Envelope12dummyMessageE
1.21	SimControl::flagValues()
1.21	SimControl::doPreActions(Star*)
1.21	SimControl::haltRequested()
1.06	Plasma::put(Particle*)
0.91	SimControl::doPostActions(Star*)
0.76	FloatParticle::die()
0.76	FloatParticle::operator<<(double)
0.61	SimControl::haltStatus()
0.61	SimControl::setPollFlag()
0.61	Queue::putTail(void*)
0.61	ListIter::operator++(int)
0.61	GenericPort::isItOutput() const
0.61	GenericPort::isItInput() const
0.45	FloatParticle::operator<<(int)
0.3	KnownBlock::findEntry(char const*, char const*)
0.3	ParticleStack::put(Particle*)
0.3	Queue::getHead()
0.3	Queue::putHead(void*)
0.3	ListIter::ListIter[in-charge](SequentialList const&)
0.15	FloatParticle::initialize()
0.15	FloatParticle::operator=(Particle const&)
0.15	Plasma::get()
0.15	Envelope::Envelope[not-in-charge](Message&)
0.15	PortHole::receiveData()
0.15	PortHole::sendData()
27.26	

Table 55 FIG runtime and events per second for no co-simulation in Chatoyant

No Co-Sim

	Chatoyant
Runtime	13.023
Events per Sec	386.712

BIBLIOGRAPHY

1. Kurzweg, T.P., Martinez, J.A., Levitan, S.P., et. al. “Modeling Optical MEM Systems,” *Journal of Modeling and Simulation of Microsystems*. Volume 2, No. 1. Pages 21-34. 2001

2. Martinez, J.A., Kurzweg, T.P., Levitan, S.P., et. al. “Mixed-Technology System-Level Simulation,” A*nalog Integrated Circuits and Signal Processing.* 29, Pages 127-149. Copyright 2001, Kluwer Academic Publishers.

3. Gerin, P., Yoo, S., et. al. “Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures”. *Proceedings of the 2001 conference on Asia South Pacific design automation*. Pages 63 -68. 2001.

4. Hübert, Heiko. “A Survey of HW/SW Cosimulation Techniques and Tools”. Thesis Work. Royal Institute of Technology, Stockholm, Sweden. June 1998.

5. Hines, K., Borriello, G. “Dynamic Communication Models in Embedded System Co-Simulation”. *Proceedings of the Design Automation Conference*. 1997. Pages 395-400.

6. Sjöholm, S., Linfd, L. “A need for Co-Simulation in ASIC-Verification”. *Proceedings of the 23rd EUROMICRO Conference*. 1997. Pages 331-335.

7. Buck, J. et al. “Ptolemy: A Framework for Simulation and Prototyping Heterogenous Systems”. *International Journal of Computer Simulation*. 1992. Pages 1 -34.

8. Bails, M., Martinez, J. Levitan, S.P., et. al. “Performance Simulation of a Microwave Micro-Electro-Mechanical System Shunt Switch Using Chatoyant”. *Proceeding of DTIP of MEMS & MOEMS*. May 2004.

9. Bhattacharyya, B. Buck, J.T., et. al. “The Almagest”. Volume 1, User’s Manual. Copyright 1990-1997. Chapters 4, 5, 7, and 12.

10. Ashenden, P.J., Peterson, G.D., Teegarden, D.A. “The System Designer’s Guide to VHDL-AMS”. Morgan Kaufmann Publishers. Copyright 2003. Pages 13-14.

11. Vee, V.Y., Hsu, W.J., “Parallel Discrete Event Simulation: A Survey”. Centre for Advanced Information Systems. Report from Nanyang Technical University. 1998

12. Banarjee, P. “Parallel Algorithms for VLSI Computer-Aided Design”. Prentice Hall. New Jersey. 1994. Chapter 7.

13. Nicol, D., Fujimoto, R. "Parallel Simulation Today". *Annals of Operations Research*. December 1994

14. Bauer, M., Wolfgang, E. "Hardware/Software Co-Simulation in a VHDL-based Test Bench Approach". *Proceedings of the Design Automation Conference*. 1997. Pages 774-779

15. Liu, J., Nicol, D.M., Tan, K. "Lock-free Scheduling of Logical Processes in Parallel Simulation". *Proceedings of the 15th workshop on Parallel and Distributed Simulation*. 2001. Pages 22-31.

16. Legedza, U., Weihl, W.E. "Reducing Synchronization Overhead in Parallel Simulation". *Proceedings of the 10th workshop on Parallel and Distributed Simulation*. 1996. Pages 86-95.

17. Preiss, B.R., Loucks, W.M., et. al. "Null Message Cancellation in Conservative Distributed Simulation". *Proc. 1991 Workshop on Parallel and Distributed Simulation*. Anaheim, CA. 1991. Pages 33-38.

18. Wood, K.R., Turner, S.J. "A generalized carrier-null method for conservative parallel simulation". *Proceedings of the eighth workshop on Parallel and distributed simulation*. 1994. Pages 50-57.

19. Levitan, S.P., et. al. "System Simulation of Mixed-Signal Multi-Domain Microsystems with Piecewise Linear Models". *Proceedings of the 2003 Nanotechnology Conference*. Volume 2, Nanotech 2003.

20. Chandy, K.M, Misra, J. "Distributed Simulation: A case study in design and verification of distributed programs". *IEEE Transactions on Software Engineering*, SE-5(5). Pages 440-452. September 1979.

21. Misra, J. "Distributed discrete event simulation". *ACM Computing Surveys*, 18(1). Pages 39-65. March 1986

22. Chandy, K.M, Misra, J. "Asynchronous distributed simulation via a sequence of parallel computations". *Communications of the ACM*, 24(11). Pages 198-205. November 1981

23. Chen, G., Szymanski, B.K. "Lookahead, Rollback and Lookback: Searching for Parallelism in Discrete Event Simulation". *Proc. Summer Computer Simulation Conference*, July 2002.

24. Reed, D.K., Levitan, S.P., Boles, J. "An Application of Parallel Discrete Event Simulation Algorithms to Mixed Domain System Simulation". *Proceedings of the Design Automation and Test Conference*. Paris, France. 2004.

25. "Transaction-based Co-Simulation with Cadence Palladium and Verisity Specman Elite". Cadence Corporation. 2003.

26. Rabaey, J. M., "Digital Integrated Circuits, A Design Perspective". Prentice Hall. 1996. Pages 111-113.

27. Smith, D.J. "HDL Chip Design". Ninth Printing. Doone Publications. July 2001.

28. Lin, Y.B., Fishwick, P.A. "Asynchronous Parallel Discrete Event Simulation". *IEEE Transactions on Systems, Man and Cybernetics*. 1995.

29. Ferscha, A. "Parallel and Distributed Simulation of Discrete Event Systems". *Handbook of Parallel and Distributed Computing*. McGraw-Hill. 1995.

30. ModelSim SE User's Manual. Version 5.8c. Published March 5th, 2004. Model Technology, a Mentor Graphics Corporation.

31. ModelSim SE Foreign Language Interface. Version 5.8c. Published March 5th, 2004. Model Technology, a Mentor Graphics Corporation.

32. Dewey, Allen. "Analysis and Design of Digital Systems with VHDL". PWS Publishing. 1997. Pages 491-492.

33. Rowson, J. "Hardware/Software Co-Simulation". *Proceedings of the Design Automation Conference*. 1994. Pages 439-440.

34. Bergeron, J. "Writing Testbenches". Kluwer Academic Publishers. 2000.

35. Evans, B.L., Kamas, A., Lee, E.A. "Design and Simulation of Heterogeneous Systems using Ptolemy". *Proceeding of the 1st Annual Conference of the Rapid Prototyping of Application Specific Signal Processors*.

36. Kahrs, M., Levitan, S.P., et. al. "System level modeling and simulation of the 10G Optoelectronic Interconnect". *IEEE Journal of Lightwave Technology*. January 2004. Volume 21, Issue #12.

37. Martinez, J.A., Kurzweg, T.P., Levitan, S.P., et. al. "System Level Simulation of Mixed-signal Multi-domain Microsystems with Piecewise Linear Behavioral Models". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. Febraury 2003, Volume 22, Issue #2. Pages 139-154

38. Verilog 1364 Hardware Description Language Standard. *IEEE Standard*. 2003.

39. Martinez, Jose A. "Piecewise Linear Simulation of Optoelectronic Devices with Application to MEMS". Thesis. University of Pittsburgh. 2000.

40. Gui, P., Kiamilev, F., et. al. "Source Synchronous Double Data Rate (DDR) Parallel Optical Interconnects". *Proceedings of InterPACK 2003*. 2003.

41. Chiarulli, D.M., "Technical White Paper: Eight Row 8x8 Crossbar Switch". University of Pittsburg, Department of Computer Science.

42. Bakos, J.D., Chiarulli, D.M. "Design of a Crossbar Switch Chip for Use in a Demonstration System of an Optoelectronic Multi-Chip Module". University of Pittsburgh, Department of Computer Science.

43. Kernighan, B.W., Ritchie, D.M. "The C Programming Language". Second Edition. Prentice Hall Publishers. 1988.

44. Stones, R,, Matthew, N. "Beginning Linux Programming". Second Edition. Wrox Press, Ltd. 1999.

45. Holzner, S. "C++ Black Book". Coriolis Publishers. 2001.