2003

A survey of hardware design verification

Liping Guo
San Jose State University

A SURVEY OF HARDWARE DESIGN VERIFICATION

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

by

Liping Guo

August 2003

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

[illegible]

Dr. Donald L. Hung

Simon Shim

Dr. Simon Shim

[illegible]

Dr. Xiao Su

APPROVED FOR THE UNIVERSITY

[illegible]

ABSTRACT

A SURVEY OF HARDWARE DESIGN VERIFICATION

by Liping Guo

The move from schematic-based design to hardware description language-based design has enabled hardware designers to easily manage the complexity of designs that were impossible to handle by manual methods. However, barely a decade after this revolutionary design methodology shift, today's hardware development industry is facing a more severe challenge. The main cause of this is that the complexity of state-of-the-art hardware devices is climbing much faster than the capacity of the techniques and tools that are used to verify them.

In the past few years, due to the effort from both academia and industry, verification techniques and tools have advanced in many ways but a fundamental breakthrough in hardware verification methodology has not yet arrived.

This thesis report presents a general survey of hardware design verification. Four important aspects of hardware design verification are covered by the survey: simulation acceleration, simulation vector generation, verification environment construction, and co-simulation in co-design.

ACKNOWLEDGEMENTS

I would first like to express my sincere appreciation to my advisor, Professor Donald L. Hung, for leading me into this interesting field and acting as my intellectual lighthouse throughout the thesis work. This thesis would not have been possible without his constant guidance and encouragement.

I would also like to thank my thesis committee members, Professor Simon Shim and Professor Xiao Su, for their valuable feedback and insightful comments.

At the final stage of my thesis work, Prasanth Kumar gave me great help by proofreading my thesis. I would like to thank him for his kindness and friendship.

Finally, I want to take this chance to thank my husband, Dr. Min Zhong. With his inexhaustible patience, he has supported me unconditionally through the ups-and-downs during my shift from Chemistry to Computer Engineering. I would not have come this far without his precious love and advice.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 The Hardware Development Crisis

In the recent decade, the advancement of semiconductor materials and processing technologies has allowed circuit designs of multi-million gates to be able to fit on a single chip. As a consequence, electronic devices are shrinking in physical size while significantly enhancing in capabilities and speed. However, this technology advancement also brought an unprecedented verification challenge for hardware development. While the increasing silicon capacity allows design complexity to grow rapidly, verification complexity is rising at a much faster pace than the staggering design complexity [1].

Unfortunately, the existing verification methodologies and tools have not kept up with the climbing verification complexity, and the situation is further aggravated by products' shortening time-to-market and consumer's insatiable demand for new product features. This troublesome reality can be vividly illustrated by the famous and notorious Pentium microprocessor's FDIV bug found in 1994: a design error not caught during the verification process forced Intel to set aside a reserve of $420 million to cover the costs, hire hundreds of customer service personnel to handle customer requests, and dedicate four fulltime employees to read Internet newsgroups and respond immediately to any postings about Intel or its products [2]. Now, a decade has past since the discovery of Pentium's FDIV bug; the inadequacy of the verification methods still severely impedes the productivity of today's hardware development industry: about 60% to 80% of the hardware design groups' effort is dedicated to verification [3] [4], and 80% of the second and subsequent chip re-spins are caused by design errors that are not captured by verification [5].

According to Joan Bartlett [6], in order to enable hardware engineers to design and implement chips of multi-million gates forecasted by Moore's Law, at least three major problems must be addressed: (1) the large volume of design detail demands a higher level of design abstraction; (2) the shortening time-to-market demands improved simulation performance and debugging techniques; and (3) the huge cost incurred by re-spin due to the un-captured bugs demands early software/hardware integration and verification in the development process.

Undeniably, in today's hardware development, verification has already become a bigger challenge than design. The pressing need for verification solutions widely affects many entities of the hardware development community. For example, the electronic design automation companies are spending more on research and development of verification tools; and in system design houses, the team of verification engineers is quickly expanding. Even though much progress has been made, finding solutions to the problems of hardware verification still has a long way to go.

1.2 Motivation and Goals

While the hardware development industry is struggling to survive the verification crisis, many engineering students haven't been aware of its severity. This worrisome situation is mainly due to three reasons: (1) the long existing negative sentiment towards verification deeply influences engineers of the next generation, and many engineering students still think that design is more important than verification and believe that the best engineers get to work on design; (2) despite the fact that formal verification has been around as an active research topic in the universities for many years, few verification-oriented courses can be seen in the engineer's curriculum and very limited verification skills are taught in the universities; (3) in the past few years, several books on hardware verification were published, but most of these books focus on specific verification topic and require considerable background knowledge to grasp the contents that the books present.

This thesis is initiated based on the awareness of the situation described above. The survey will contribute in three ways: (1) for engineering students who haven't been exposed to hardware verification, it can serve as an introductory reading material to hardware design verification; (2) for engineers who worked in industry and had experiences in hardware verification, it provides updated information on the topics that are covered by the survey; and (3) for the author herself, it establishes the foundation for her continued research in this field.

1.3 Scope of the Thesis

During the course of a hardware development, at least three levels of validation must be undertaken to ensure the correctness of the final product [7]. These three levels of validation are design verification, implementation verification, and manufacturing testing.

Design verification is a process of ensuring that a design exhibits intended behavior [8]. There are two broad approaches to hardware design verification:

simulation-based methods and formal methods. Because design verification targets the initial hardware description language (HDL) description, usually the first description of a design, it is of crucial importance in the design process.

Implementation verification checks if the design is correctly implemented with respect to its specification. Once an initial HDL description of a design is validated through extensive simulation or formal property verification, it proceeds through a varied set of optimization and transformation operations. Implementation verification is to check whether or not the optimized and transformed design is functionally equivalent to the original one. Equivalence checking tools are often used in various stages of the design cycle to verify equivalence of different implementations of the same design.

Manufacturing testing detects manufacturing defects in a fabricated chip. At this stage, the design description has been validated and the implementation of the design has also been verified. So, the purpose of manufacturing testing is to capture errors resulting from flaws in the chip fabrication process, rather than design errors.

Among these three levels of validation, *design verification* is the main concern of this thesis. Particularly, the survey is centered on simulation-based design verification approach. Since it is impossible to touch every aspect of hardware *design verification*, the thesis focuses on four important subjects, namely, simulation acceleration, simulation-vector generation, verification environment construction, and hardware-software co-simulation. For each subject, the basic background information is introduced; the current research progress and industry trend are described; the research examples selected from various publications are presented; and the updated information on available tools in that area is provided.

1.4 Thesis Outline

This thesis report is organized in seven chapters:

Chapter 1 describes today's hardware development crisis and explains where this thesis fit in.

Chapter 2 contains background information on hardware design verification.

Chapter 3 presents methods that are used to accelerate the simulation process. Among these methods, hardware-accelerated simulation techniques are discussed in detail.

Chapter 4 surveys techniques for simulation vector generation. Several semi-formal vector generation methods are introduced.

Chapter 5 discusses verification environment construction. A collection of hardware verification languages is introduced, and topics, such as layered testbench architecture, testbench automation, and verification intellectual property (VIP), are also included.

Chapter 6 describes the hardware-software co-simulation in co-design domain. Background information on co-design is provided; co-simulation techniques and available tools are presented.

Chapter 7 summarizes the survey.

2 BACKGROUND

2.1 Introduction

To cope with the verification challenge in hardware design, tremendous effort has been made, by hardware development industry and research institutions all over the world, to find solutions to the problems of hardware verification. In general, the available solutions to hardware design verification can be categorized into two broad approaches: formal verification and simulation-based verification [9]. Formal verification methods, such as theorem proving, model checking, equivalence checking, etc., attempt to mathematically prove the correctness or incorrectness of the designed systems. On the other hand, simulation-based verification models the design, in either software or hardware, and tries to detect the design errors by applying tests to the modeled design and observing its behavior.

In the past, formal hardware verification has been around mainly as an academic exercise. Its steep learning curve, low automation level, and inability of handling designs of large size have prohibited formal verification from being adopted in the hardware design flow. Only in the recent two to three years, have commercial formal verification tools started emerging into the market and the employment of formal methods in industry begun making promising progress. Meanwhile, as an unsolved problem, formal verification will remain as a hot academic research topic for some time. In contrast, simulation-based verification, which holds an imperative status, has been the industry's mainstream approach to hardware design verification for many years. However, as the size of a design grows the effectiveness of the traditional software-only simulation decays rapidly.

This chapter contains necessary background information on both simulation-based verification and formal verification. In section 2.2, three major approaches of simulation-based verification are described, and the tools commonly used in each approach are presented; in section 2.3, formal verification methods and tools are briefly introduced.

2.2 Simulation-based Verification

As described in section 2.1, simulation-based verification attempts to detect the design errors by modeling the design in either software or hardware, applying tests to the modeled design, and observing its faulty behavior. There are three major approaches used in today's simulation-based verification [10]: software-only simulation, hardware-accelerated simulation, and in-circuit emulation. The later two methods, hardware-accelerated simulation and in-circuit emulation, are also referred as hardware-based simulation.

2.2.1 Software-only Simulation

- Advantages and Disadvantages

In *software-only simulation*, the hardware design along with its testbench is completely modeled in software. Besides being cost-effective, this approach also offers another two benefits [11]: (1) it provides the possibility of observing and controlling the internal signals of the design under test (DUT); and (2) it allows performing simulation at the early stage of designs, regardless the model being synthesizable or non-synthesizable. Because of its beneficial features, software-only simulation is by far the most popular functional verification method employed in hardware development industry.

The software-only simulation has some drawbacks. Firstly, it requires long computation times and capacities, and the situation gets worse as the design complexity increases. Secondly, although many fault coverage metrics have been developed, there is still no practical way to directly associate simulation coverage with the confidence in a design that is gained through simulation (in fact for many complex designs total coverage is impossible to reach). As a result, when the simulation of a design can be stopped is often dictated by the time-to-market, not the simulation coverage. Thirdly, software-only simulation normally does not consider the physical environment that the designed circuit is employed in. In such a situation, it is impossible to observe the circuit's real performance because the circuit is isolated from its real environment during simulation.

- Simulators

Table 1 contains a collection of commercially available simulators. A detailed description for each included simulator can be found at the web link provided under "comments."

Table 1: Commercially available simulators

Simulator	**Vendor**	**Comments**
Incisive™	Cadence	Single-kernel architecture natively supports Verilog, VHDL, SystemC, SystemC Verification library (SCV), and PSL/Sugar assertions. http://www.cadence.com/products/incisive_unified_simulator.html
NC-VHDL	Cadence	A VHDL simulator. http://www.cadence.com/products/ncvhdl.html

NC-Verilog	Cadence	A compiled Verilog simulatorand good for gate level simulation. http://www.cadence.com/products/ncverilog.html
NC-SystemC®	Cadence	First commercial implementation of the SystemC Verification Library. http://www.cadence.com/products/ncsystemc.html
Verilog-XL	Cadence	The most standard simulator in the market, as this is the sign off simulator. http://www.enee.umd.edu/class/enee408c/Verilog-XL/
VCS™	Synopsys Inc.	A Verilog simulator. http://www.synopsys.com/products/simulation/simulation.html
Scirocco™	Synopsys Inc.	A VHDL simulator. http://www.synopsys.com/products/simulation/scirocco/scirocco.html
VCS-MX	Synopsys Inc.	Supports mixed-HDL simulation. http://www.synopsys.com/products/simulation/vcs-sci/vcs-sci.html
Scirocco-MX	Synopsys Inc.	Supports mixed-HDL simulation. http://www.synopsys.com/products/simulation/vcs-sci/mixed_hdl_ds.html
Finsim	Fintronic USA Inc.	A Verilog simulator. Supports the entire Verilog HDL including all behavioral, gate and switch level constructs, user defined primitives, specify blocks, system tasks and functions, PLI 1.0, VCD and SDF; runs on Linux, Windows, and Solaris; supports compiled, interpreted and any mixture of compiled and interpreted simulation. http://www.fintronic.com/frame_products.html
Modelsim	Model Technology *A Mentor Graphics Company*	Supports VHDL and Verilog with Single Kernel Simulation; good for block level verification. http://www.model.com/products/default.asp
Smash	Dolphin Integration	Mixed-signal simulation performed with a mixed-signal netlist; Single-engine simulation for both the analog and logic parts; Complete language support and mixity with SPICE, VHDL, VDHL-AMS, VERILOG, C and soon VERILOG-AMS. http://www.dolphin.fr/medal/smash/smash_overview.html

2.2.2 Hardware-accelerated Simulation

- Advantages and Disadvantages

The *Hardware-accelerated simulation* method is employed to speed up the simulation process. In hardware-accelerated simulation, the DUT is mapped to field programmable gate arrays (FPGAs) or special-purpose computing engines, and a testbench is used to provide stimulus (simulation vectors) to the DUT. Depending on the implementation of the testbench, either leaving the software-modeled testbench in the simulation environment or loading the synthesized testbench onto the accelerator, hardware-accelerated simulation can be 10 to 100,000 times faster than software-only simulation [12][13].

Table 2: Commercially available accelerators

Accelerator	**Vendor**	**Comments**
Hammer™	Tharas Systems	ASIC-based hardware accelerator for Verilog, VHDL and mixed language simulations; offers debugging capability comparable to that of software simulators. http://www.tharas.com/products/index.html
Xcite®	Axis Systems	Based on Axis' patented ReConfigurable Computing (RCC) technology, delivers simulation and acceleration in a single system using one design database; offers simulation performance of up to 100K cycles/second on a design capacity of up to 10M ASIC gates; provides RTL simulation acceleration. http://www.axiscorp.com/products/xcite.html
Cobalt^{plus}	Quickturn *A Cadence Company*	A custom processor-based, compiled code logic emulation system; offers a capacity of up to 20 million emulation gates, 2 Gbytes DRAM and 256 Kbytes SRAM memory in a single chassis; comes in two configurations: The CE series (CE1000-CE8000) with one million gate granularity and the CL series (CL10000-CL20000) with 2.5 million gate granularity. http://www.quickturn.com/products/cobaltplus_data_sheet.htm

Unfortunately, hardware-accelerated simulation is much more costly than software-only simulation. It requires a design or a portion of a design be modeled in a synthesizable way before conducting simulation. Moreover, the DUT is still isolated from its real application environment during simulation since the design is exercised through stimulus provided by the testbench, not the real life data.

- Accelerators

Table 2 contains information on a collection of commercially available accelerators. Features of each accelerator provided in Table 2 are not comprehensive. A detailed product description can be found at the web link listed under "comments" for each accelerator.

2.2.3 In-circuit Emulation

- Advantages and Disadvantages

Despite its high cost, *in-circuit emulation* offers the highest run-time performance among the three simulation-based verification approaches. Different from hardware-accelerated simulation, with in-circuit emulation, the stimulus to the DUT is not provided by testbench; instead, it comes from the live electrical connections consistent with the application environment [13]. In-circuit emulation allows the DUT to be verified within its real application environment, and thus makes hardware-software co-verification and system-level verification possible.

However, in-circuit emulation can only provide a limited visibility of the internal signals of the DUT [9], and, like hardware-accelerated simulation, it also requires the design to be synthesizable before verification can be performed.

- Emulators

A detailed product description can be found at the web link listed under "comments" for each accelerator. Also, emulators annotated as *hybrid product* can be used as either accelerators or emulators.

Table 3: Commercially available emulators

Emulator	**Vendor**	**Comments**
System Explorer	Aptix Corp	Provides Internet-based verification service http://www.aptix.com/products/product_overview.htm
CelaroPRO	Mentor Graphics	Supports up to 64 independent clock domains; offers multi-user shared capabilities and remote network access with queuing support. http://www.mentor.com/celaro/
Mercury^{Plus}	Quickturn *A Cadence Company*	Provides capacity of up to 20 million ASIC gates and a memory system with up to 2 gigabytes of memory that supports large testbenches and software code for High-Performance regression testing. http://www.quickturn.com/products/mercuryplusspec.htm
Xtreme *Hybrid Product*	Axis Systems	Speeds emulation performance up to 500K cycles per second Supports three modes of in-circuit verification (ICV): in-circuit simulation (ICS), in-circuit acceleration (ICA), and in-circuit emulation (ICE). http://www.axiscorp.com/products/xtreme.html
Vstation-5MX *Hybrid Product*	Mentor Graphics (Ikos)	Uses patented VirtualWires™ technology to ensure that the same database can be implemented in every replicate VStation hardware solution. Up to 15 million usable ASIC gates (4.5 million on a VStation-5Mx) ; performance in the MHz range http://www.mentor.com/vstation/vstation5mx.html
Palladium *Hybrid Product*	Quickturn *A Cadence Company*	Provides up to 100x to 10,000x RTL performance; provides simulation acceleration and in-circuit emulation in a single system; supports multiple users and remote access; maximize efficiency with fast compiles on a single workstation. http://www.cadence.com/products/palladium_new.html

2.3 Formal Verification

Simulation-based verification methods have a major drawback: they cannot fully verify every aspect of a system's functionality due to the fact that exhaustively simulating a design is infeasible, unless the design is an extremely simple one. Consequently, only the portion of the possible behaviors of a design is simulated; and serious design errors often remain undetected and later cause catastrophic problems.

The evolution of the application of formal methods in hardware verification directly results from the inability of the simulation-based methods in fully verifying a system's functionality. As a complementary approach, formal methods use mathematical means to prove that a design is correct without applying huge sets of test vectors. In [14], Mcfarland vividly described the difference between formal verification and simulation-based verification: "The difference between formal verification and simulation is similar to the difference between deriving laws in physics from first principles and performing experiments."

Since formal verification is not the main concern of this survey, this section only provides a brief introduction to formal hardware verification techniques and tools. However, there exist a large number of publications addressing this topic. In particular, several surveys (e.g., [7], [15], [16], [17], [18], [19], and [20]) provide comprehensive coverage of both theoretical and practical aspects of formal hardware verification. Also, a new book titled "Introduction to Formal Hardware Verification" authored by Thomas Kropf was released recently. As claimed in its description, this advanced textbook presents an almost complete overview of existing techniques for formal hardware verification.

2.3.1 Formal Methods

Ideally, a formal method consists of a formal language, tools, and a proof system, which can be used to specify and verify systems [21]. A formal language refers to a language where every well-formed statement has a mathematically defined meaning; and tools are used to help the designers to describe systems and requirements in the formal language; finally, a proof system provides facilities to be used in reasoning about statements in the formal language.

In a design process, there are two main aspects to the application of formal methods [15][21]: one is the *specification oriented* formal framework, for example temporal logic, predicate logic, etc., which provides a rich and mathematically precise language that is used to specify intended properties of a design; the other is the *verification oriented* formal techniques and tools, which are used to reason about the relationship between a specification and its corresponding implementation. There exist quite a few formal techniques, such as automata-theoretic technique, automated theorem proving, model checking, equivalence checking, symbolic trajectory evaluation, as well as many hybrid ones that integrate these formal techniques with a hope of improving verification performance [22] [23] [24] [25].

Having a specification for a system written in a formal way brings many benefits to a design project [21]. The formal frame work forces the desired properties of a system to be specified in a precise description, which eliminates ambiguity that often exists in the informal specifications, especially the ones written in natural languages, such as English; also, when a specification is written in a formal language, it becomes possible to reason about whether or not the specification matches the true intention of the designers. On the other hand, the *verification oriented* formal techniques and tools attempt to make sure that an implementation confirms to its corresponding specification [15]. In order to check this conformance, formal descriptions for both the specification and the implementation must be created first, and formal tools can then be used to establish the conformance.

It is important to note that formal methods are applied to the models of systems, not to the real systems themselves [21]. In other words, the usefulness of the verification results obtained via formal methods heavily relies on the correctness of the models and the specifications. For instance, if a model or a specification fails to accurately capture the behaviors of a system, then proofs acquired through formal methods are meaningless. This fact simply demonstrates how critical it is to create an authentic model/specification for a system. Unfortunately, obtaining such a model or a specification is still a challenge [26].

2.3.2 Formal Methods in Hardware Verification

Applications of formal methods in hardware verification domain are often formulated in two forms [27]. One form is the so called *property check* or *property verification*, which is concerned with properties and a model of a design; and verification on this respect attempts to show that all of the system's possible behaviors captured in the model satisfy the temporal properties of its specification [15]. The other hardware formal verification form is known as the *implementation check* or *implementation verification*, which addresses relationship between two models of a system. In this case, verification is to show that each possible behavior of a system's implementation (a model) is consistent with some behavior of its high-level specification (a high-level model of the system).

In practice, *property verification* and *implementation verification* are often used in conjunction [15]. The property verification is usually conducted first to prove that a high-level model of a system satisfies a set of desired temporal properties defined by its specification; then, a lower level model with more detailed implementation is developed, and implementation verification is carried out to verify if the lower level model is an implementation of the high-level model. The later process is iterated as the design development proceeds: a series of models are developed, and each of them is an implementation of the model at the next higher level.

Three formal hardware verification methods are commonly used: theorem proving, model checking, and equivalence checking. Among these three approaches, theorem proving and model checking deal with the problem of *property verification*; and equivalence checking addresses the *implementation verification* issue.

Theorem proving is one of the earliest formal approaches used in hardware verification [7]. In this approach, the implementation (e.g. a high-level model of a system) and the specification of a system are both described in some kind of formal logic, such as higher-order logic, first-order logic, etc.; and the relationship between them is regarded as a theorem in the formal logic [16]. The goal of the proving process is to establish that the theorem, which expresses the relationship between the implementation and the specification of a system, is a logical truth in the system.

Theorem proving approach is structural rather than behavioral [7]. This distinctive feature allows a circuit to be described hierarchically, where a component defined at one level in the hierarchy serves as an interconnection of components defined at lower levels.

As a general approach, theorem proving places no restriction on applications. It also offers powerful logic expressiveness and allows a circuit's behaviors to be described and related at many different levels of abstraction. However, theorem proving is an interactive method. Verification techniques based on theorem proving demand considerable effort on users' part in developing specifications and guiding the theorem proved through all the lemmas. Unfortunately, this demand often exceeds the capabilities of industrial hardware designers.

Model checking, opposite to theorem proving, is behavioral rather than structural, since only the behavior of a system is checked to satisfy certain properties [7]. In this approach, properties of a system are specified in the language of temporal logic, and the system is modeled as finite state machine. The goal of the checking process is to establish whether properties hold by exhaustively searching through the state space of the model [21].

Model checking is one of the most widely used automatic methods of verifying hardware [21]. The major strength of it, besides being an automated method, is its ability to produce counterexamples that can be used as a critical aid to debugging [27]. However, because the state space of any non-trivial system is extremely large, especially when the system consists of many asynchronous communicating state machines, model checking faces the so called "state explosion" problem, which limits its applications to the large systems and at low abstraction level [7]. Techniques, such as symbolic methods [28], are developed by various groups to alleviate the problems of state explosion, and some of them show promising results.

Equivalence Checking uses mathematical techniques to determine whether one design representation is functionally equivalent to another. This formal approach is often applied at the last stages of a design cycle to ensure that the final design implementation does what the register transfer-level (RTL) code specifies [29].

The main strength of equivalence checking is that the checking process can be fully automated. Also, equivalence checking can be used as a good replacement of regression tests, since it is most effective in verifying the functional equivalence of slightly different designs [27], for example, the slightly changes introduced at the last stages of a design to optimize performance.

Because verifying a system via equivalence checking relies on the comparison between two representations of the system, it is helpful only if one of the representations of the system is correct. This correct representation is also referred as the so called "golden reference." Unfortunately, equivalence checking is not able to verify the correctness of the golden reference itself, and other means must be utilized to ensure the correctness of the golden reference.

2.3.3 Formal Hardware Verification Tools

Although formally verifying an entire design is not generally possible in today's hardware development, formal verification techniques are making their ways into the design flows for complex systems [30]. This promising progress, at least to some extent, should attribute to the recent proliferation of commercial verification tools.

A few years ago, formal verification tools were used merely within the academic scope, and the deployment of formal tools in industry was often seen as discouraging. This situation was mainly caused by the weakness of these formal tools: they usually provided limited capacity and poor usability, and many hardware engineers found them hard to use and difficult to integrate into the existing design flow.

While simulation-based verification techniques are losing their effectiveness in verifying designs of large size, more and more hardware design companies, including large systems houses and semiconductor suppliers, start seriously considering deploying formal techniques in their hardware verification processes. The demands for better formal verification tools encouraged EDA vendors, from well-established companies like Cadence and Mentor Graphics to some start-ups, to invest more in research and development on formal verification tools. As a result, a number of formal verification tools, such as equivalence checker and property checker, are commercially available today; and formal verification tools are gaining more acceptance and support in the hardware development industry. This claim can be well supported by the list of customers of a small EDA company called *Verplex* (Milpitas, CA), which produces both equivalence checker and property checker. According to Michael Chang [29], president and chief executive officer of Verplex Systems, Inc., in the first half of 1999, Verplex only had seven customers, but today 150 different electronics companies worldwide are using Verplex' formal verification tools.

Table 4: Commercially available property checkers, Lars Philipson 2001

Product	**Released & Current Version**	**Vendor & Web Site**	**Strong Points According to Vender**
@Verifier	2001 2.1	@HDL www.athdl.com	1. Automatic Property Extraction; 2. Automatic Testbench generation; 3. Multiple clock domins; 4. Incremental checking; 5. Bounded and unbounded checking
BlackTie	2000 2.10	Verplex Systems www.verplex.com	1. Ease of use; 2. Automatic checks; 3. Capacity & performance
Design Verity-Check	2000 1.1	Veritable www.veritable.com	1. Ease of use; 2. Capacity & performance
Formal Check	1997 3.1	Cadence Design Systems www.cadence.com	1. Capacity; 2. Large installed base; 3. Sophisticated Methods; 4. Ease of use
Formal Model Checker	1997 2001.2	Avanti www.avanticorp.com	1. Mature technology; 2. Supports safety, liveness, and fairness
Improve-HDL	2001 1.2 beta	TNI-Valiosys www.valiosys.com	1. Capacity; 2. Accepts non-synthesizable HDL; 3. Properties & constraints in a single language
Solidify	1999 2.5	Averant www.averant.com	1. Practicality; 2. Language easy to use; 3. Capacity& performance; 4. Incremental use
Verix	2000 3.0	Real Intent www.realintent.com	1. Ease of use; 2. Hierarchical verification; 3. High error coverage; 4. Synthesis of simulation checkers; 5. Multiple formal engines

Lars Philipson, a professor at Lunds Tekniska Hogskola University in Lund, Sweden, has done an excellent survey on formal model checking and equivalence checking tools [26]. In the survey, Philipson collected almost all the current commercially available equivalence checkers and model checkers that specifically target at hardware verification. The total of ten EDA vendors were identified, and four equivalence checkers and eight property checkers were included in Philipson's survey. Table 4 contains information about the eight property checkers (model checkers); and Table 5 provides information of the four equivalence checkers. All the information shown in Table 4 and Table 5 were extracted from Philipson's survey [26], which was originally published in Swedish in the *Elektronik I Norden* magazine in November 2001, and later the English-language version was made available to *EEdesign*.

Table 5: Commercially available equivalence checkers, Lars Philipson 2001

Product	Released & Current Version	Vendor & Web Site	Strong Points According to Vendor
Conformal LEC	1998 3.0	Verplex Systems www.verplex.com	1. Capacity & performance; 2. Usability; 3. Integrated transistor extraction
Design Verifyer	1993 2001.2	Avanti www.avanticorp.com	1. Accuracy; 2. Large database of customer examples; 3. Large selection of available options; 4. Speed and ease of use
Formality	1998 2001.06	Synopsys www.synopsys.com	1. Capacity performance; 2. Debugging environment; 3. Mixed language support; 4. Uses Verilog simulation libraries
FormalPro	2000 3.1	Mentor Graphics www.mentorg.com	1. Capacity; 2. Debug features; 3. Mixed HDL; 4. Automation; 5. Save session

3 SIMULATION ACCELERATION

3.1 Introduction

The advancement of the CMOS technology has drastically changed the way that the electronic products are designed, verified, and manufactured. On one hand, engineers are able to quickly produce smaller, faster, and more sophisticated products; on the other hand, the task of verifying a design becomes unprecedented heavy and complex. This increased verification load can be simply illustrated by looking at the number of simulation vectors needed to simulate a design. Comparing to the designs of a few years ago, the number of simulation vectors required to verify today's hardware designs has more than doubled [13].

Obviously, exercising a design with more simulation vectors requires more simulation time. This nature of the simulation-based verification approach seriously affects its effectiveness in handling large designs and meeting today's short time-to-market requirement. Before formal methods becoming the industry's mainstream verification means, hardware design verification has to largely depend on simulation. In order to make simulation-based approach more capable in verifying designs of large sizes, solutions to speedup simulation must be found.

This chapter surveys techniques used to speedup the simulation process. Section 3.2 discusses simulation acceleration by using faster simulators; Section 3.3 introduces the use of simulation server farms; Section 3.4 describes approaches based on simulation-emulation co-operation; and Section 3.5 briefly illustrates the issues of creating the verification environment.

3.2 Using Faster Simulators

Hardware simulation can be conducted in two ways: (1) simulating a design without considering timing and (2) simulating the models including timing [31]. The tradeoffs between these two approaches are performance and accuracy.

Traditional event-driven simulators calculate every active signal for every device it propagates through during a clock cycle. By examining detailed timing information, event-driven simulators provide higher accuracy in verification and therefore the rich functionality of the DUT can be verified through this type of simulation [31]. However, due to the high signal activity within a clock cycle, the performance of event-driven simulation, in terms of speed, is relatively poor.

Cycle-based simulator, on the other hand, provides an inexpensive method of accelerating functional simulation [32]. Typically, cycle-based simulators are five to ten times faster than conventional event-driven simulators. Such performance improvement is achieved through the following: (1) cycle-based simulators compute only two logic states: 1s and 0s; (2) cycle-based simulators calculate the results only at the clock edges; and (3) the inter-phase timing is ignored during simulation. In addition to performance improvement, because of the elimination of the inter-cycle calculations, cycle-based simulators also reduce the memory utilization to about one-fifth up to one-third comparing to the event-driven simulators.

Cycle-based simulators have their limitations. For example, cycle-based simulators are mainly used for synchronous designs; when components of an asynchronous design are simulated using cycle-based simulators, the performance penalty is high. However, as the complexity of hardware verification increases exponentially, correctly employing the cycle-based simulation definitely offers a relief in terms of speed and memory usage.

In the past few years, the performance of software simulators has improved constantly: compiled simulators increases simulation speed by more than ten times over the interpreted simulators [33] [34]; and cycle-based simulators boost simulation performance by avoiding detailed event processing overhead. Unfortunately, simulators' performance improvement cannot cope with the climbing complexity of the verification tasks. To further improve the speed and increase the capacity of functional simulation, other methods must be utilized.

3.3 Using Simulation Server Farms

In the past, when hardware design complexity reached 100k gates, traditional interactive simulation was replaced by batch-level simulation to achieve better performance [35]. Now, as the design complexity of a typical electronic system often exceeds a million gates, many hardware design companies are turning to simulation server farms for massive simulation performance gain.

Basically, a simulation server farm consists of a group of computers that run multiple simulations of a design simultaneously. Comparing to single computer simulation, the use of simulation server farms can increase simulation throughput by several orders of magnitude [36]. Among various approaches to speeding up simulation, server farm technology has become a proven and reliable solution for simulating large and complex designs. The server farm technology offers many benefits [35]: it provides more simulation capacity; it allows the utilization of cost effective hardware and software resources; it offers a way of maximizing the utilization of workstations and licenses; and it is scalable.

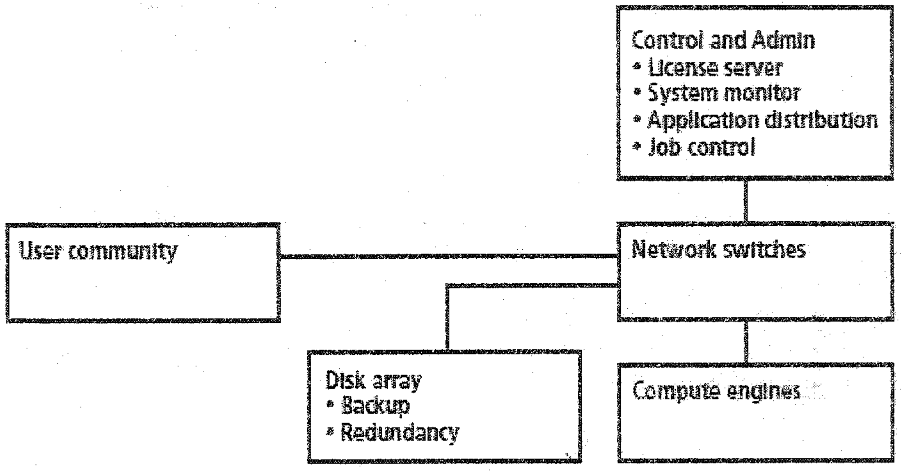


Figure 1: Simulation server farm topology

A typical simulation sever farm, at its most basic level, is a collection of hardware that is managed by some specialized software [37]. The collection of hardware normally includes compute engines, storage servers, administrative servers, and networks [37]. Figure 1 (from [37]) shows an example of a simulation farm topology. Although a simulation server farm can start from any size or price spectrum, issues, such as verification plan, equipment selection, installation, and configuration, must be properly addressed to ensure the success of a newly built server farm. Professional help can also be acquired from some EDA vendors. For instance, Cadence offers services of setting up simulation farms, which include planning, equipment selection, and installation. More information on design and implementation of a simulation farm can be found in Cadence's white paper [37]; and Synopsys' white paper [35] also contains information on hardware selection and setup for a simulation farm.

3.4 Using Simulation-Emulation Co-operation

3.4.1 Motivation

Both software-only simulation and in-circuit emulation have advantages and disadvantages. Software-only simulation offers good observability and controllability, and it can be used to simulate both synthesizable and non-synthesizable design models. In contrast, in-circuit emulation provides good performance in terms of speed, and it can have the DUT simulated within its real application environment.

The motivation of simulation-emulation cooperation, also termed as "hardware-accelerated simulation," stems from the desire of combining the advantages of both software-only simulation and in-circuit emulation.

3.4.2 Examples

In [11], Siavash et al. reported a simulation-emulation co-operation method for Verilog and VHDL models, and their experimental results show that the proposed method can significantly reduce the simulation time. The reported method utilizes a general simulator as well as a general emulator: a PCI-based PLDA board is used for the emulation part and the ModelSim simulator (Version 5.5 a) is used for the simulation part. The overall co-operation environment is illustrated by Figure 2.

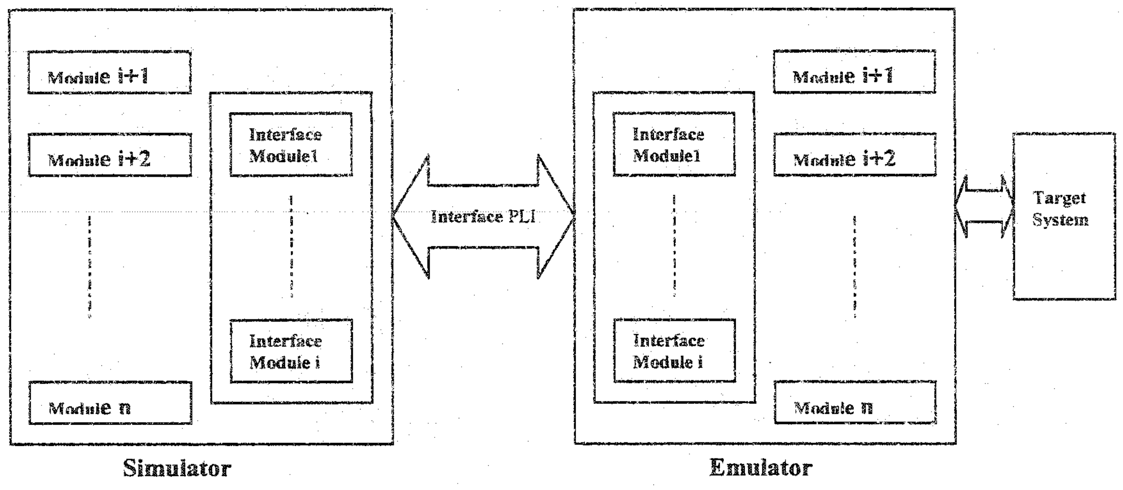


Figure 2: Siavash et al.'s method of simulation-emulation co-operation

The simulator (ModelSim 5.5a) resides on a host computer, which is connected to the emulator (the PLDA board) via a PCI (Peripheral Component Interconnection) expansion slot. The FPGA mounted on the PLDA board can be configured through the PCI bus, and the PCI bus then serves as the communication channel between the configured FPGA and the host computer. On the emulator side, the user-defined logic (portion of the DUT) communicates with the PCI bus through the PCI core (the *Interface Modules*), which handles all bus events; on the simulator side, the simulator communicates with the FPGA chip by means of the PLI routines (the *Interface Modules*), which access the PLDA board via memory-mapped I/O technique.

To use the simulation environment presented in [11], the system description, either in VHDL or Verilog, needs to be partitioned into the simulation part and the emulation part. Both parts are then compiled or compiled and configured into sub-circuits for simulation and emulation respectively. During the simulation process, the simulator controls and coordinates the activities of the emulators through the PLI routines, and the corresponding PLI routine is invoked by the simulator when information from the emulator is needed.

In [38], Canellas and Moreno proposed a simulation-emulation co-operation method of for VHDL models. Unlike [11], no actual physical emulator is used in Canellas and Moreno's co-operation environment; instead, another simulator is utilized to act as a logical emulator. Figure 3 shows the basic idea of the simulation environment.

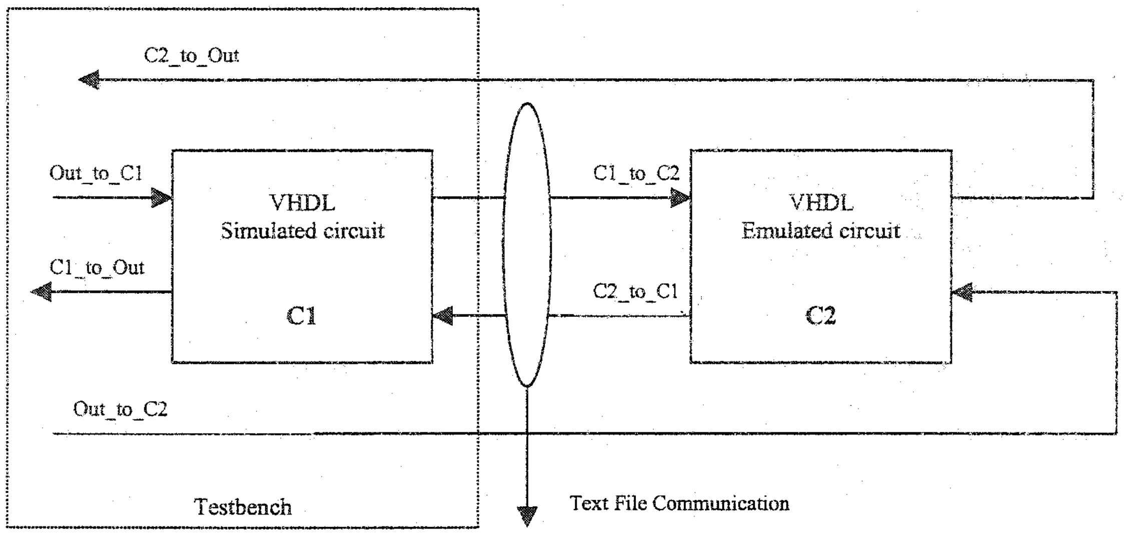


Figure 3: Canellas and Moreno's co-simulation/emulation

The to-be-verified design is split into two parts: VHDL simulated circuit and VHDL emulated circuit. The simulator and the emulator run on different computers. The communication between the simulator and the emulator is achieved by means of text files; in this case, the TEXTIO package available in the standard VHDL library is used as the communication text files.

Since no actual physical emulator is employed, this method is relatively cost-effective; however, the use of text files as a way of inter-processes communication can be a bottleneck of this approach, and thus the communication overhead prevents the method from being applicable to large designs.

In [39], a hardware acceleration scheme for functional logic simulation is presented by Cadambi et al. In this work, a standard "off-the-shelf" PCI-board (ADC_RC1000 from AlphaData) with a single FPGA is used as the hardware accelerator. Instead of synthesizing the DUT directly onto the FPGA, an intermediate simulation processor, called SimPLE, is mapped onto the FPGA, which acts as an execution engine for the netlist during the simulation. The overall acceleration system is illustrated in Figure 4.

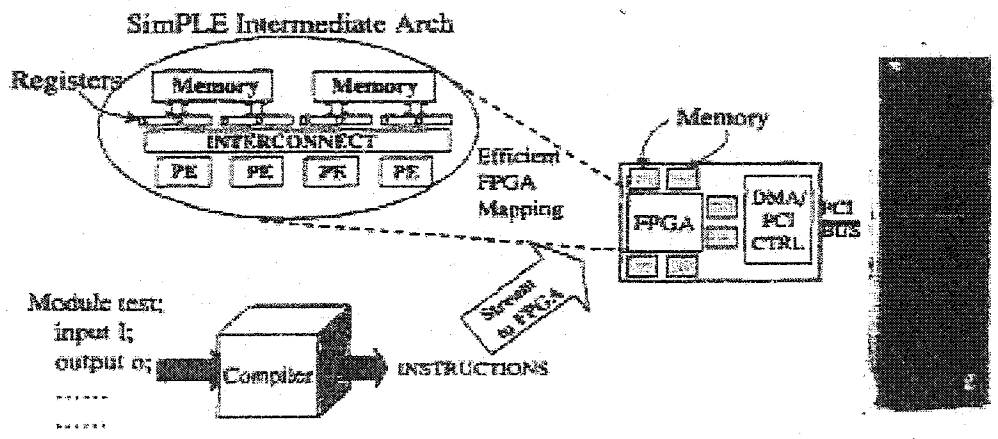


Figure 4: Cadambi et al.'s acceleration system

The DUT is compiled into VLIW-type instructions by a fast SimPLE compiler, and each instruction represents a slice of the netlist of the DUT. The compiled instructions along with a set of simulation vectors are then transfer to the on-board memory via direct memory access (DMA). For each simulation vector, SimPLE executes all instructions to assure that the entire netlist is tested, and the simulation result is stored back to the on-board memory. After all the simulation vectors are exercised, the simulation results are sent back from the board memory to the host, again via DMA.

The whole simulation process is controlled by the host through application program interface (API). Scalability can be obtained at the price of sacrificing performance, by breaking up the instructions into smaller portions and transfers them separately to the on-board memory.

The authors reported that the proposed scheme can obtain speedups of up to 2000x over zero-delay event-driven simulation and up to 1000x over cycle-based simulation on benchmarks and industrial circuits. The authors attributed the simulation speed gain to the following factors: the SimPLE's parallel architecture, the large number of registers and memory in SimPLE, the high bandwidth between the FPGA and on-board memory, and the high clock speed of the FPGA.

In [40], Kirovski, Potkonjak, and Guerra reported a cut-based functional debugging paradigm that leverages the advantages of both emulation and simulation. In this approach, test vectors are applied to the DUT by emulation tool to achieve simulation speed improvement. When the design error is detected during emulation, the computation can be switched over to simulation tool for full design visibility and controllability. The execution can be rolled-back to any arbitrary instance in run time, which eases debugging process. The authors claimed that, with a low hardware overhead, the proposed approach along with its accompanying algorithms demonstrated effectiveness when used on a set of benchmark designs.

There are many other publications on simulation-emulation co-operation, and here only some representative ones are presented. As analyzed before, the purpose of this co-operation is to take advantages of both simulation and emulation. Simulation provides good controllability and observability for signals in the design, but its slow speed prevents it from effectively handling large designs; in-circuit emulation, on the contrary, can achieve high execution speed, but its poor controllability and observability makes debugging much more difficult. The desired solution would be a simulation method that offers high speed and, at the same time, maintains good controllability and observability.

3.5 Using Rapid Construction of Verification Environment

A verification environment is an infrastructure for simulation-based verification. Before conducting simulation using any simulation oriented techniques, a verification environment must be properly constructed. The way that a verification environment is architected has an immediate impact on simulation efficiency, and, more importantly, it affects the possibility of reusing the same verification environment, or certain components of it, for verification of other designs.

However, creating a verification environment is a time-consuming process. When a new product's time-to-market is shrinking in a never-ending manner, the time spent on creating a verification environment and the time actually used on simulating a design have to be well-balanced, so as to achieve satisfactory verification productivity. Among the endeavor s of accelerating the overall simulation-based verification process, searching for means to reduce the time consumed on constructing a verification environment is absolutely an imperative one.

In recent years, issues of improving the effectiveness of a verification environment and reducing the time spent on constructing it have been intensively addressed. For example, verification specific languages, such as Vera and the e-language, were developed to remedy the deficiency of using hardware description languages to handle verification tasks; the increased automation level has ever changed the traditional concept of a testbench; and the desire of reusing verification components became a reality with the introduction of verification intellectual property (VIP). All those topics are discussed separately in Chapter 5: Verification Environment Construction.

4 SIMULATION-VECTOR GENERATION

4.1 Introduction

Simulation-based verification attempts to capture design errors by applying simulation vectors (test patterns) to the design-under-test (DUT) and observing its faulty behavior.

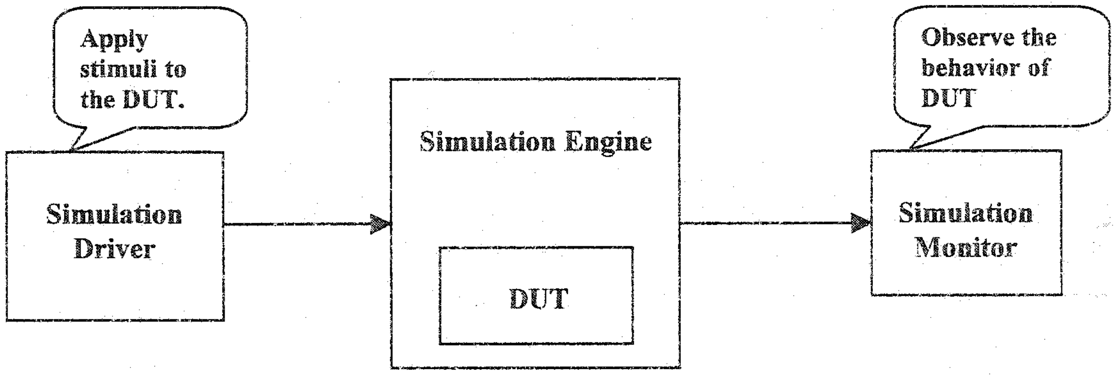


Figure 5: Conventional simulation scheme

Simulation vectors are simply input vectors to the DUT that causes the presence of a design error to be observable during simulation. Because of the increasing size of the design and the growing demand for bug-free product, the amount of the simulation vectors required to verify a design is growing larger and larger. The magnitude of the testing space has reached the level that it is impossible to probe every single point in it due to the time and resource constraints. Therefore, to reduce simulation time without sacrificing the verification quality, the most effective simulation vectors must be obtained such that the least amount of simulation vectors can be used to uncover the greatest number of design errors. Usually, the effectiveness of a set of simulation vectors is measured by coverage metrics, such as code-based metrics, functionality-based metrics, spec-based metrics, observability-based coverage, etc. [41].

An ideal approach to verifying a design would be *exhaustive testing*, in which all possible input vectors and their combinations are applied to the DUT. Theoretically, this method can achieve the highest design error coverage; realistically, however, it is infeasible to exercise all input combinations unless the DUT is a combinatorial logic of very small size, which is often not the case in reality.

In industry, the following two types of simulation-vector generation techniques form the backbone of the traditional simulation-based verification methodology [42]: *directed testing* and *random testing*. In directed testing, the simulation vectors are manually created by hardware designers, and they aim at verifying the important functionalities of a design. Since the simulation vectors are hand-crafted, some hard-to-detected exceptional (the "corner") cases can be purposely targeted. Unfortunately, directed testing suffers from some major drawbacks: (1) the simulation-vector generation process is far from automation, which means experienced designers have to dedicate a considerable amount of time in writing tests; (2) the completeness of the manually generated tests is almost impossible to acquire as it is difficult to think up and write test vectors that would simulate every aspect of a design; (3) the errors introduced by misinterpretation of a design specification or incorrect specification itself are likely to escape, since the designers are the same group of people who interpret the specification and later write the tests. In random testing, the simulation vectors are randomly generated. Comparing to the directed testing, random testing is a simpler and cheaper testing method. The problem of the randomly generated vectors is their inability of detecting redundant fault [43] and their poor error coverage for most sequential circuits [44].

Simulation vector plays a critical role in simulation-based verification. The quality of simulation vectors have a direct impact on simulation performance: good simulation vectors are short so that simulation time can be reduced; good simulation vectors have high error coverage so less bugs will be missed. The simulation vectors' importance drives many hardware engineers and researchers to seek for better ways to generate test vectors of high quality.

This chapter highlights some of the research work that has been done on simulation-vector generation. Section 4.2 introduces vector generation techniques that borrow test sets used for physical fault testing, and section 4.3 describes some semi-formal methods for simulation-vector generation.

4.2 Borrowing Test Sets from Physical Fault Testing

4.2.1 Motivation

Physical faults usually refer to manufacturing defects in digital systems that are introduced because of the imperfection of the chip fabrication processes. Comparing to functional verification, physical fault testing, or chip testing, is one of the more successfully tackled problems in today's VLSI chip development [45]. Often, physical faults are modeled as logical faults, which, in turn, can represent many different physical defects, such as shorts, opens, bridges, etc. The frequently used fault model is the so called "single stuck-line (SSL)" model, in which a single interconnection line is permanently stuck at one logic value: *stuck-at-zero* or *stuck-at-one*. Test generation algorithms for such fault models, like the D-algorithm, the path oriented decision making (PODEM) algorithm, etc., are well-studied; and automatic test pattern generation (ATPG) tools, such as GENTEST, ESSENTIAL, FASTEST, etc., are commercially available. So, the questions are: whether the test generation techniques for physical fault testing can be borrowed for simulation vector generation; and if they can, how to make use of the borrowed test vectors in tackling design errors during functional simulation.

The similarity between hardware design verification and physical fault testing has inspired some investigators to adopt physical fault testing techniques in generating test sets for design errors [8] [46] [47] [48] [49]. However, the gap in abstraction level between the implementation and the specification often imposes difficulties in mapping empirical design errors to physical fault models [50]; and the test generation for large sequential circuits, such as pipelined microprocessors, is still a to-be-solved problem.

4.2.2 Examples

In [8] and [46], Hussain Al-Asaad and John P. Hayes proposed a model-based automated design validation scheme for gate-level combinational and sequential circuits, which borrows method from test generation for physical faults. In this work, commonly seen design errors that cause malfunction of logic circuits are classified into five types: the gate substitution error (GSE), the gate count error (GCE), the input count error (ICE), the wrong input error (WIE), and the latch count error (LCE). For each type of these gate-level design errors, its unique characteristic is analyzed and the corresponding detection requirement is derived; then the design error is carefully mapped into single stuck-at line (SSL) faults; finally, a standard automatic test pattern generation (ATPG) program and simulation tools are employed to generate test vectors, which will be utilized as simulation vectors in design verification process. The critical step of this method is the mapping from some design error model to the single stuck-at line fault model. This process includes the modification of the netlist of the to-be-tested design (or equivalent description) and the injection of a set of predefined SSL faults.

The experiments on the benchmark circuits demonstrate that the generated test sets via proposed approach are small and have high error coverage. Al-Asaad and Hayes also pointed out that a design passes the tests is guaranteed to be correct with respect to the modeled faults only, and this limitation is due to the fact that the complete set of the design error model is unknown.

In [47], Chen et al. studied the application of universal test set (UTS) to design verification. The UTS, based on the unate function theory [51], has been studied by many researchers [52] [53]. The results based on those studies have indicated that the UTS, generated from the functional specification, can be used to detect single and multiple stuck-at faults. Chen et al. analyzed the relationship between the design error models (e.g. the missing wire error, the extra wire error, etc.) and the stuck-at fault models. The analysis shows that the test set used to detect stuck-at faults detects most of the design errors. This indication motivated Chen et al. to use UTS as test vectors for design error detection. The reported experiment results show that the application of UTS to design verification is efficient and memory-saving.

4.3 Simulation-vector Generation via Semi-formal Methods

4.3.1 Motivation

With the rapidly growing verification complexity, many engineers and researchers try to combine the advantages of formal verification and the traditional simulation-based verification together to validate large designs [42] [54] [55] [56] [57] [58] [59]. Many of the studies attempt to make the existing simulation-based verification more formal by using coverage metrics as heuristic measures to quantify the completeness of verification and to guide the generation of input vectors.

4.3.2 Example: Coverage Guided Random-simulation-vector Generation

The traditional random testing randomly selects as many test vectors as possible with the hope of achieving high simulation coverage. However, the metrics used to measure the coverage of a set of simulation vectors and the method employed to generate those vectors are often disconnected [54]. This disconnection unavoidably leads to poor simulation coverage even a large number of random simulation vectors have been applied to the DUT. Many researchers have tried to solve this problem by integrating coverage metrics into the simulation vector generation process [42] [54] [55] [57]. For instance, [42] uses trajectory graph as a coverage metric to quantify the test coverage of a set of simulation vectors; and [57] presents a method using genetic algorithms to guide random simulation. The similarity among those studies is that they all try to use certain guidelines to direct the random simulation so that to achieve higher simulation coverage. Below, Tasiran and Fallah's work is elaborated to show the essence of the guided random simulation.

In [55], Serdar Tasiran and Farzan Fallah presented a simulation-based semi-formal verification method for sequential circuits that are described at the register-transfer level. In their work, the generation of the simulation vectors is guided by the observability-based coverage analysis. Figure 6 depicts their biased-random simulation process.

The coverage metric used to direct the simulation-vector generation is the so called "tag coverage metric." In this coverage metric, a code segment is considered covered only when it is exercised and affects an observed node of the circuit during simulation. Apparently, tag coverage is superior to code-based coverage for its paying special attention to the case where a design error is exercised but its effects never get propagated to the observed nodes.

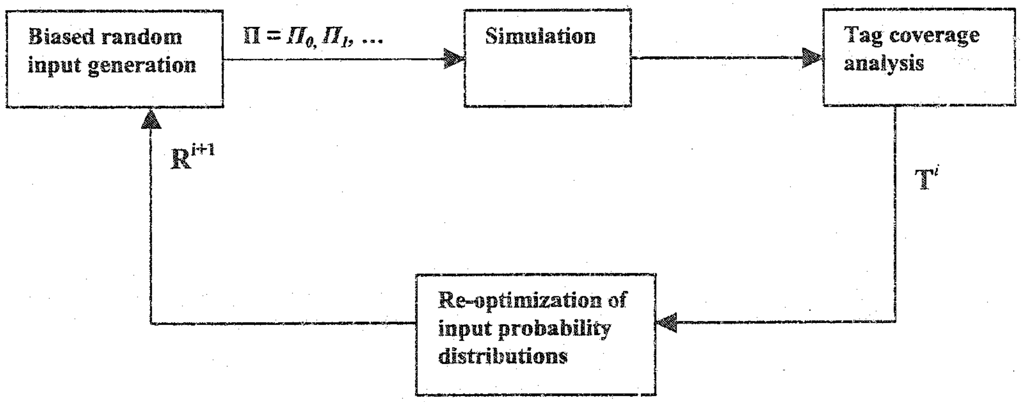

Biased random input generation
$\Pi = \Pi_0, \Pi_1, \ldots$
Simulation
Tag coverage analysis
R^{i+1}
T^i
Re-optimization of input probability distributions

Figure 6: Tasiran and Fallah's biased-random simulation with coverage feedback

Within the framework of the proposed method, for a given DUT and a set of tags, the goal is to determine the probability distributions (PD) for the primary inputs of the circuit in conjunction with possible input vectors. The probability distribution function R controls the selection of input vectors to the circuit during simulation runs: $R_i(v) = \alpha$, which means input i is assigned to v with probability α. As illustrated in Figure 6, the observability-based coverage metric (here the tag coverage metric) is used to identify portions of the DUT not exercised by the previous simulation run. The PD optimization algorithm then selects the R^{i+1} based on the computation of the "merit functions," which analyzes the tags to be covered, the DUT, and the probability distribution function and estimates the number of tags that have a high likelihood of being detected during the simulation for each given R. Finally, the R^{i+1} is used for biased random input pattern generation that targets at the non-covered portions of the circuit. A simulation run using R^{i+1} is performed until the tag coverage stops to improve, and the PD optimization is performed again to target the remaining set of tags. The whole process repeats to the point that no more tags are covered.

The reported experimental results indicate that the presented approach can achieve better tag coverage than uniform PDs in much fewer simulation cycles for some circuits, but there are some circuits that the uniform PDs give poor coverage and the proposed approach also failed to improve the coverage considerably. Tasiran and Fallah also suggested that the biased random simulation be complemented with other deterministic methods to achieve overall good verification performance.

4.3.2 Example: Coverage Guided Simulation Vector Transformation

In [56], C. Norris Ip described another semi-formal technique to guide simulation vectors generation. In this technique, the concept of abstract state exploration history, which was often used for reachability analysis in formal verification, is introduced to a simulation environment. Moreover, a test stimulus transformation method, which can transform the existing test suite to a desired new test suite on-the-fly, is also presented.

A state exploration history is simply a summary of what simulation vectors have been applied to the DUT and what design states have been exercised. During simulation, this state exploration history is maintained in a very abstract fashion and used to provide the basis for test stimulus transformation. Figure 7 illustrates the test stimulus transformation process proposed by Norris Ip.

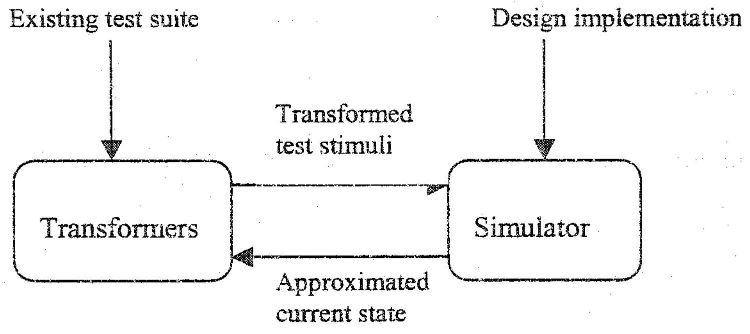


Figure 7: C. Norris Ip's test stimulus transformation

In a typical traditional simulation scheme, after a simulation environment exercises a design description by applying stimuli (test vectors) from the testbench, some coverage tool is used to measure the coverage, and then new test vectors are manually written to target at the uncovered portion of the design. Different from the traditional simulation scheme, Norris Ip's approach tries to automate the generation of test vector. It utilizes a dynamically collected state history as a concrete coverage metric to guide the stimuli transformation for obtaining more effective simulation vectors.

As depicted in Figure 7, a set of transformers and a simulator are concurrently running during simulation: the simulator collects the state exploration history while performing its regular simulation task; while the transformers perform test suite transformation for covering more unexplored states, based on the analysis of the history information provided by the simulator and the examination of the test suite for the current simulation step.

Experiments on two practical designs, the DASH cache coherence protocol and MPEG2 decoder, show that the reported method can increase simulation coverage and reduce simulation time. In theory, if the state exploration history maintains a complete history of state exploration, 100% simulation coverage could be achieved for a DUT; however, the *state explosion problem* makes it infeasible to keep a complete state exploration history. Thus, although the method removes lots of the redundant work caused by ineffective simulation vectors, the full simulation coverage is not guaranteed. The uniqueness of this method, as claimed in [56], is that it uses a transformation framework instead of a test vector generation framework to produce simulation vectors; therefore the existing test suite and the simulation technology can be reused.

4.3.3 Example: Other Semi-formal Approaches

Normally, the effectiveness of the simulation vectors is measured by some coverage metrics, but coverage metrics cannot always correlate with error coverage because the relationship between them has not been well understood. In [59], Gupta et al.

present their work on generating a property-specific testbench for guided simulation. Although their work focuses on the automatic generation of testbenches instead of simulation vectors, a vector generator along with a checker is embedded in the testbench obtained. The vector generation constraints are determined based on the analysis of the design and the properties being checked, rather than coverage measures.

There is also a line of work that uses symbolic methods to achieve effective simulation [60] [61]. For example, Geist et al. reported a semi-formal method for direct test vector generation by using symbolic techniques [60]. In their method, a combination of model checking techniques and symbolic simulation is used to generate test vectors, and the vector-generation constraints are derived based on the transition coverage.and some temporal properties in a finite state machine.

5 VERIFICATION ENVIRONMENT CONSTRUCTION

5.1 Introduction

Verification environment construction is a fundamental process in a hardware design cycle. This process consumes a huge portion of verification time and has a major influence on the overall verification productivity. In recent years, new languages, ideas, and tools that address the efficiency of constructing the verification environment, are proliferating. For instance, quite a few languages have been developed for hardware verification; the concept of intellectual properties (VIPs) has been brought into the verification community, and the EDA companies has begun to offer testbench automation tools that automate many verification tasks that were done manually in the past.

This chapter collects information on the construction of verification environment. Section 5.2 contains descriptions of a line of hardware verification languages; Section 5.3 briefs the status of the industry-standard hardware verification language; Section 5.4 introduces layered testbench architecture; and Section 5.5 discusses verification intellectual property.

5.2 Hardware Verification Languages (HVL)

5.2.1 Motivation

The longing for a new verification language stems from the realization of the existing hardware description languages' limitation. Currently, the hardware design is still dominated by the use of Verilog and VHDL; and these two languages were created at the time when the hardware engineers knew exactly what gates they wanted and where [62]. Inherently, the existing HDLs are more suitable for register transfer level (RTL) design; they lack abstraction and are not very expressive with respect to the testing constructs. As design moves towards system level and verification complexity gets more and more severe, the existing HDLs can no longer efficiently handle verification tasks to meet the demands for higher productivity and better quality. The deployment of high-level languages designated for verification purpose becomes irresistible.

According to Synopsys' white paper entitled "OpenVera Technology Backgrounder" [63], a desirable high-level verification language would be the one that meets the following requirements: (1) It should support the modeling of testbench functionality at a high level of abstraction; it must be able to specify the stimulus (test vectors), compute the expected response of the device to the applied stimulus, and model the test fixtures that allow tests to be applied and results to be checked. (2) It should support directed test, random test, constrained -random test, and the random pattern generator must allow the constraints to be specified in a compact, declarative way. (3) It should support the specification of metrics to measure that, to what extent the verification goals have been met; and the language should allow these metrics to be queried dynamically so that the stimulus generation can be adjusted to maximize test effectiveness. (4) It should also allow the user to specify connections to HDLs and C in a convenient way.

Broadly speaking, there are two ways of forming a hardware verification language: (1) by extending the existing languages' capabilities, for example, SystemVerilog extends the power of Verilog; or (2) by introducing a new verification language, such as Vera. Since many EDA vendors tend to use their own proprietary languages within their dedicated environments, there exist far too many hardware verification languages. A collection of hardware verification languages that briefly described below is not intended to serve as a complete list of the existing HVLs. However, the HVLs included in the collection are believed to be the most popular ones. The list is inspired by [64] with a few more languages added to it, and the placement of the languages is according to the alphabetical order.

5.2.2 Examples of the Existing HVLs

5.2.2.1 DGL

DGL, stands for "data generation language," was originally designed to generate functional level tests for VLSI designs. Although DGL was based on a probabilistic context free grammar, it provides several features for generating non-context free languages to accommodate the fact that many tests contain context sensitive data or data that is difficult to describe using a context free grammar. The DGL was designed to facilitate the construction of data generators, and the DGL compiler can be used to create a data-generator once the format of the test data had been described in DGL. Tests generation using DGL can be either systematic or random, or combination of both [64]. According to the DGL reference manual [65] authored by the creator of DGL, Peter M. Maurer, DGL is still under development and will probably remain so for some time and mainly used as a research tool to support research activities in VLSI testing and design verification.

5.2.2.3 e-language

The e-language [66], introduced by Verisity (www.verisity.com), is created especially for verification purpose. It is built on object-oriented programming paradigm with additional constructs provided for specification and verification. Verisity has launched its LicenseE program for open licensing of the e verification language [67] with a hope of making e-language the de facto standard verification language in hardware industry. According to [68], industry leading electronics companies including ARM, Cisco Systems, etc. have joined the LicenseE program to help drive the e language towards open public standardization.

5.2.2.4 Jeda

Jeda [69], a new functional verification language, was first developed at Juniper Network, Inc and then moved to Jeda Technologies, Inc. The language is designed for modeling and verifying hardware design. Jeda supports flexible and dynamic concurrent programming with time/cycle concept; it also provides various system classes, such as semaphore and events, to allow the efficient construction of dynamic concurrent systems. In addition, Jeda supports object oriented programming, and the newly released Jeda 3.0.0 version comes with the feature that supports a new programming paradigm - aspect oriented programming. The syntax of Jeda is based on C and Verilog with a few concepts borrowed from C++, Java, and Perl. Jeda can either run with Verilog as the user PLI code or be used as a standalone simulator. Jeda Technologies, Inc (www.jedatechnologies.com) provides commercial support for Jeda Verification System.

5.2.2.5 Libero

Libero [70], written by Pieter Hintjens, is a free Programmer's Tool and Code Generator that uses a finite-state machine as the underlying model. Users define the high-level logic of a problem as a diagram, and Libero generates the code to implement the written logic. Libero can generate code in C, C++, Java, Perl, Awk, 80x86 assembler, COBOL, MS Visual Basic, MS Test Basic, UNIX C Shell, UNIX Korn Shell, UNIX Bourne Shell, GNU Bash Shell, Rexx, PL/SQL, and PHP.

5.2.2.6 Murphi

Murphi [71], a free protocol description language and a verifier for finite state concurrent systems, was originally designed and implemented by David L. Dill, Andreas Drexler, Alan J. Hu, and C. Han Yang; and later, it has been rewritten, enriched, and maintained by Ralph Melton, Seungjoon Park, C. Norris Ip, and Denis Leroy. Currently, Murphi is maintained by Ulrich Stern (http://sprout.stanford.edu/uli/) and Norris Ip (http://sprout.stanford.edu/ip/).

Murphi is a protocol description language based on a collection of guarded commands – conditions or action rules. The Murphi verifier, which is based on explicit state enumeration, works by explicitly generating states and storing them in a harsh table [72]. The algorithmic techniques in Murphi aim at exploring a given state space in the most efficient manner to allow verification of large protocols. Murphi also provides several special techniques for reducing the number of reachable states while guaranteeing that protocol errors will still be detected [73].

5.2.2.7 RAVE

RAVE [74] is short for "Reuse Architecture for Verification," and it is a proprietary verification language from Forte Design Systems (www.forteds.com). RAVE provides verification-specific command set, which allows user to easily create data sources that model complex real-world requirements. The language is designed to support efficient generation and manipulation of large data sets, extensive random value generation, creation of complex transaction flows, computation of expected values, and reactive data generation and execution control for complete functional coverage. RAVE has been used to support Forte Design's verification suite - Quickbench Sequencer, and its automated validation features play an important role in this product, especially in runtime testbench control and direct information feedback to the designer.

5.2.2.8 Solidify

Solidify [75] [76] is a proprietary hardware property language from Averant (www.averant.com). It is a HDL-like language that can be used to describe design behavior. As a general-purpose property verification language, Solidify can be used to express temporal relationships, sequence of events, and verification loops; and it also supports user-defined variables, recursive macros, procedures, and functions. Averant's *Solidify*, an HDL (Verilog and VHDL) static functional verification tool that exhaustively verifies properties of Verilog or VHDL designs, is based on this proprietary language.

5.2.2.9 Spin

Spin [77] is a software tool that is widely used for the formal verification of distributed software systems. It was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. Spin targets software verification, not hardware verification. Spin offers three usage modes: it can be used as a simulator that allows rapid prototyping with a random, guided, or interactive simulations; it can be used as an exhaustive verifier, which is capable of rigorously proving the validity of specified requirements; it can also be used as a proof approximation system that validates large protocol systems with maximal coverage of the state space. More features about Spin can be found at Spin Home Page [77].

PROMELA, Spin's input language, is a non-deterministic language that is loosely based on Dijkstra's guarded command language notation and the notation for I/O operations from Hoare's CSP language.

5.2.2.10 SUPERLOG

Superlog, developed by Co-Design Automation Inc., is a superset of Verilog with the addition of C programming, system and verification capability. The language was designed to be used as a single language for four purposes: system specification, hardware description, design verification, and programming [78]. Superlog's creator Co-Design Automation Inc has tried to move a step toward the eventual standardization of Superlog [79], and it has donated what it calls the extended synthesizable subset (ESS) of Superlog to the Accellera standards organization [80]. In August 2002, Co-Design

Automation Inc was acquired by Synopsis Inc, the EDA leader, who intends to combine Co-Design's Superlog, with Synopsys' own Verilog simulator [81].

5.2.2.11 SystemC

SystemC [82] [83], a product of Synopsis Inc., is a standard design and verification language that can be used to describe hardware/software systems at multiple levels of abstraction. It is built in C++, an object oriented language, and it offers high abstraction modeling and fast simulation performance [6]. The formation of SystemC Verification Library (SCV) has extended SystemC's capability to verification domain (please cross refer to the description on TestBuilder).

SystemC is maintained by Open SystemC Initiative (OSCI), which is a non-profit organization composed of major EDA and IP companies, universities, and individuals that contributes to and governs SystemC's development and distribution. More information about SystemC and OSCI can be found at the web site: www.systemc.org.

5.2.2.12 SystemVerilog

SystemVerilog, owned by Accellera, is an Accellera (www.accellera.org) approved standard hardware design and verification language (HDVL) [84]. The language is an evolution of IEEE 1364 Verilog standards, and it extends Verilog into the high-level design and verification domain. The intention of the language is to eliminate many of Verilog's past limitations and offer a unified language for both design and verification. For design purpose, SystemVerilog provides high-level design constructs for concise design; to support verification, SystemVerilog 3.1 has added assertions and testbench automation capabilities [85]. The introduction of assertions into SystemVerilog allows designers to define design properties more concisely, and the application of these assertions to smaller modules or sub-blocks can reduce certain verification load on unit level testbenches [85]. SystemVerilog's newly added testbench automation features can be applied to create directed, random, pseudo-random tests and many other aspects to automate testbenches.

5.2.2.13 SpecC

SpecC language is a system-level language that was specifically developed to address the challenges of system-on-chip (SoC) design that involves both software and hardware. It was developed in Gajski's group at University of California, Irvine (http://www.cecs.uci.edu/~specc/), and it has been proposed as a standard system-level language for adoption in industry and academia [86]. The standardization of the language is promoted by STOC, SpecC Technology Open Consortium (www.specc.org).

SpecC is based on the ANSI-C software programming language, and it is a superset of C language [87]. The language development has focused on adding keywords to the basic C language to support hardware description at a high level as a basis for synthesis [62]. The SpecC allows the system functionality to be specified in a clear and precise way and the obtained specification can be used for simulation and as the input to the synthesis in the SpecC design methodology [87]. SpecC is also based on program state machine (PSM) model of computation to model a system at different levels of abstraction [88]. Although SpecC is backed by more than 30 companies and 30 universities worldwide [88], the precise meaning of the execution semantics is still under development [86]. More information about SpecC can be found at SpecC Technology Open consortium's web site: http://www.specc.org.

5.2.2.14 TestBuilder

TestBuilder, developed by Cadence Design Systems Inc., is a powerful testbench authoring language that extends C++ for hardware verification [64]. The open source TestBuilder provides HDL signal class, data structures, concurrency, transaction-based verification, constrained randomization and temporal check API [64].

In the effort on unifying SystemC and TestBuilder into one standard language for design and verification at both the system level and RTL level [89], Cadence Design Systems, Inc. proposed Open SystemC Initiative (OSCI) to extend SystemC by leveraging open source TestBuilder's verification functionality. The proposal was granted by OSCI, and the specification is called SystemC Verification Library (SCV), which provides a verification foundation for SystemC [89]. The SCV includes the following features: randomization facility, random constraint facility, verification models, transaction recording support streams, transaction recording facility, smart data objects, and full power and expressiveness of C++. More information on TestBuilder and SCV can be found at Cadence Verification Extension (http://www.testbulder.net/).

5.2.2.15 OpenVera

OpenVera is an open-source hardware verification language. It is developed specifically for functional verification. The language's predecessor is Synopsys' (www.synopsys.com) proprietary language Vera. Vera is built on object-orientated programming principle. It extends the capabilities of general-purpose languages, such as C++, by adding constructs needed for effective design verification. Vera supports complex data structures, built-in data types for verification, powerful set constructs for defining constraints [90], and many other essential features; and the language, developed especially for testbench writing, allows users to build testbenches at a higher level of abstraction than HDLs (Verilog and VHDL) [91].

In attempting to forge an industry-standard testbench generation language, Synopsys Inc opened its proprietary language –Vera- as verification language standard in April 2001 [91]. Synopsys also has launched the Vera Open Source Initiative for third-party vendors. The open-source license of Vera is administered by the Synopsys-backed OpenVera Group [92]. The latest development of OpenVera can be found at the web site: www.open-vera.com.

5.3 The Reality of Industry-Standard Verification Languages

Although tools, like generators, predictors, checkers, etc., that target at the hardware verification bottleneck are continuously emerging into the market, none of them is able to independently fulfill the verification task in a comprehensive fashion – no one tool is best at everything. To gain enough confidence in the designs without missing the time-critical opportunity in the marketplace, engineers have to rely on multiple tools, either developed in-house or purchased from the EDA vendors, to verify their designs. This practice made integration of tools essential.

Furthermore, grasping a verification tool often involves learning a new design/verification language. The EDA companies had historically developed and marketed tools based on their own closed, proprietary design languages, which unavoidably imposes unnecessary learning curves on engineers as one need to know "the language" in order to use "the tool." This extra learning load made the deployment of new tools much more difficult than it should have been. Currently, there are two endeavors trying to unload the language learning burden: one is to develop tools that support multiple hardware verification languages, and another is to form an industry-standard hardware verification language.

To address this "language explosion" problem and reduce engineer's learning load, some EDA companies are developing tools that support mixed languages. For example, Mentor Graphics' (www.mentor.com) simulator *ModelSim* supports VHDL, Verilog, C/C++, SystemC, and SystemVerilog [5]; Avery Design Systems, Inc. (www.avery-design.com) has built a transaction-based verification test development system called *TestWizard* that eliminates the need of learning a new language by allowing the tests be written in Verilog, VHDL, C/C++, and Perl [93]; Forte Design Systems' popular verification results analysis product *Perspective* added support for OpenVera last year in addition to QuickBench, C/C++, and HDLs [94].

It is commonly recognized by the EDA industry that tools based on proprietary languages may result in dead ends because they only foster small and fragmented markets [5]. Meanwhile, the hardware design community is hunger for interoperability solutions: the engineers are putting too much effort on procuring and developing translators, syntax checkers, and other inefficient workarounds to make sure that the new tools interoperate smoothly with their own internal solutions. Therefore, forming an industry-standard hardware verification language is endorsed by both the EDA vendors and the hardware design community.

Open Source is a popular method for creating and promoting standards in industry [95]. As described in section 5.2, languages like SystemC, e, SpecC, Vera, etc. are all being pushed for standard to certain extent through the Open Source model. The intention of those endeavors is to develop a complete solution to verification bottleneck around an open, non-proprietary hardware verification language.

The form of any standard is an outcome of users' demands combined with tool provider's efforts. As one looks back at the success of the HDL standards (Verilog and VHDL), the similar trends have been seen and are growing stronger in the process of forming industry-standard hardware verification languages. For instance, Synopsys released its commercial SystemC simulator, *CoCentric System Studio*, which provides an advanced SystemC design and verification solution [96]; Forte Design Systems' popular verification result analysis product *Perspective* added support for OpenVera last year [94]. It will probably take some time for the standard HVLs to reach the success level that HDL standards had reached years ago; however, much progress has been made and the future of the standard HVLs is promising.

5.4 Layered Testbench Architecture

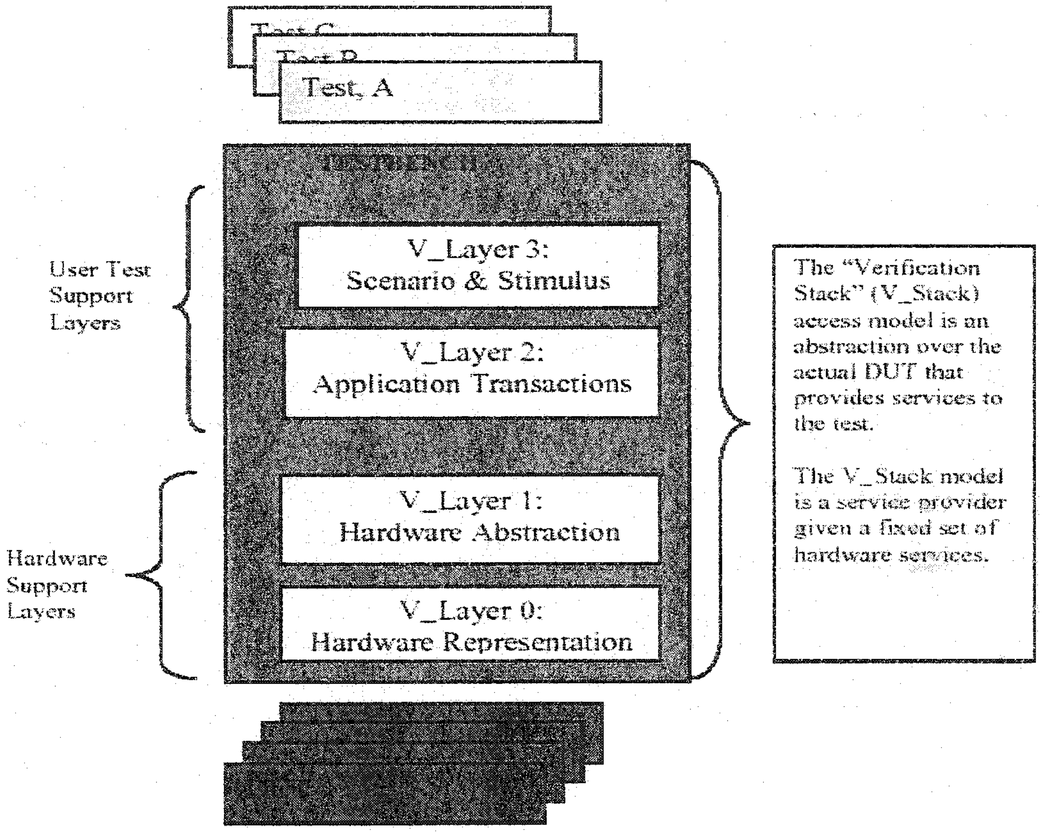
5.4.1 Motivation

In simulation-based verification, tests and DUTs are connected by testbenches. The construction of testbenches is an important task in simulation-based verification, and the verification effort expended on testbench development has made considerable impact on the overall verification quality and cost.

Traditionally, a testbench, in general, consists of a stimulus driver (a.k.a. test vector generator), an output monitor, and some mechanism of checking off completed tests and results. The shortening verification cycle demands more traditional manually performed verification tasks be handed over to testbench, which brings the so called "Testbench Automation." Basically, testbench automation tools, such as Verisity's *Specman Elite*, Synopsys' *Vera*, Avery's *TestWizard*, etc., attempt to address the functional verification bottleneck by automating the verification process. For example, *Specman Elite* automates test generation and uses functional coverage analysis to ensure the completeness of the verification [97]. Furthermore, in order to minimize the verification cost and increase verification productivity, issues such as reducing the effort on test development and debugging, enabling reuse of tests for sub-system level testing in system-level testing, maximizing the reuse of the testbench across multiple projects, etc., must be seriously considered. All those factors necessitate improved testbench architecture. Two layered testbench architectures are introduced below.

5.4.2 Examples

In [98], Mohammed Hawana and Rindert Schutten, from Synopsys Professional Services, developed a systematic approach for testbench design, which targets functional verification of complex multi-function system on a chip (SoC). In their approach, the testbech is modeled as the so called "Verification Stack (V_Stack)." The V_Stack consists of four layers: the hardware representation layer (V_Layer0), the hardware abstraction layer (V_Layer1), the application transaction layer (V_Layer2), and the scenario and stimulus layer (V_Layer3). Figure 8 (from [98]) shows layered testbench architecture.



Test C
Test B
Test A
TESTBENCH
User Test Support Layers
V_Layer 3: Scenario & Stimulus
V_Layer 2: Application Transactions
Hardware Support Layers
V_Layer 1: Hardware Abstraction
V_Layer 0: Hardware Representation
The "Verification Stack" (V_Stack) access model is an abstraction over the actual DUT that provides services to the test.
The V_Stack model is a service provider given a fixed set of hardware services.

Figure 8: Hawana and Schutten's layered solution

The lower layer provides a certain set of services to the upper layer while shielding it from the lower-level details. Each layer in the stack performs clearly defined tasks: V_Layer 0 provides signal-level connectivity into the physical representation of the DUT; V_Layer 1 provides a bus-abstraction view of the hardware to ensure that the bus transactions issued by upper V_Layers will reach the DUT, regardless of how the DUT is represented; V_Layer 2 provides the abstraction needed to carry out the operations of the SoC from the application point of view; V_Layer 3, the highest layer, enables stimulus generation and provides high-level interface to configure the DUT, testbench, and the rest.

By using a layered approach, the proposed testbench architecture increases the abstraction level on which the tests are written, which consequently reduces the effort required to write a single test; it also uses self-checking techniques and advocates using pseudo-random stimulus generation combined with functional coverage to reduce the number of tests that are needed to excise the DUT; moreover, it allows the testbench to be used for multiple configurations, which enables the sub-system level tests to be re-used in the system level context.

In this article, Mohammed Hawana and Rindert Schutten also reported the successful use of the proposed approach in two designs: a Security Processor and a Fault-tolerant PCIX to PCIX bridge chipset. The success demonstrates that the proposed layered approach can significantly reduce the effort on test creation and debugging task. The detailed explanation along with a real example can be found in Synopsys' white paper: Testbench Design, a Systematic Approach [98].

In [99], Bernd Stohr et al. presented FlexBench, a complete framework for SoC verification at the Module and SoC level. The proposed verification framework is based on a layered architecture depicted in Figure 9.

As shown in Figure 9, the proposed architecture is comprised of three layers: the stimulus layer, the integration layer, and the service layer. The stimulus layer is the "user interface" to the DUT, which provides a transaction level interface to the integration layer;

the integration layer is implemented as a set of Drivers (D) and Monitors (M), which performs actual driving and monitoring of the DUT pins; the service layer provides some system services and connects the stimulus layer and the integration layer. FlexBench is an example that shows how the verification productivity can be significantly increased by raising the abstraction level from pin-level to the higher level.

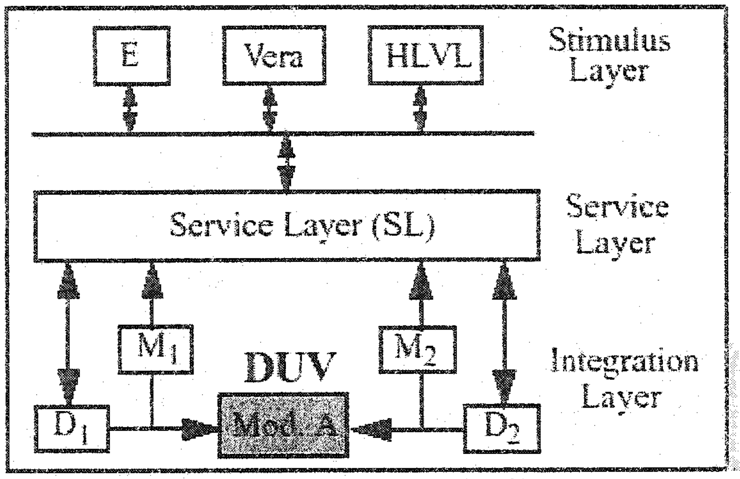


Figure 9: Bernd Stohr et al.'s FlexBench architecture

5.5 Verification Intellectual Property (VIP)

The use and reuse of intellectual property (IP) in the hardware design community has become a common practice. The motivation behind it is the desire to close the gap between the cost and the time-to-market. As design verification gets more and more complex and time-consuming, there has arisen a pressing need for standalone, pre-verified, and built-in verification infrastructure, which can be easily plugged in the simulation-based verification tests [100]. This pressing need brings the arrival of the so called "Verification Intellectual Property (VIP)."

The purpose of verification IP is to aid engineers in the task of validating the functionality of a design by accelerating the development of a complete verification environment. Since the verification IP components are pre-verified to the standard protocols and contain the necessary infrastructure for testbench generation and checking mechanisms [100], the use of VIP components can reduce the time spent on building the verification environment, thus cut down the time to the first test.

Table 6: OpenVera verification IP solutions

Company	**OpenVera Verification IP**
ControlNet India	IEEE 1394, TCP/IP Stack
GDA Technology	HyperTransport
HCL Technologies	I2C
Integnology	Smart Card Interface
nSys	SIEEE 1284, UART
Qualis Design	Ethernet 10/100, Ethernet 10/100/1G, Ethernet 10G, PCI-X, PCI, PCI Express Base, PCI Express AS, 802.11b, ARM AMBA AHB, USB 1.1, USB 2.0
Synopsys, Inc.	AMBA AHB, AMBA APB, USB, Ethernet 10/100/1000, IEEE 1394, PCI/PCIx, SONET, SDH, ATM, IP, PDH

In the VIP development community, Synopsys, a world leading EDA company, plays an initiative role. In September 2001, Synopsys launched the OpenVera Catalyst

Program, which creates a broad network of verification companies with expert tool and methodology knowledge in leading verification solutions [101]. One important mission of the participating members of the Catalyst Program is to create and commercialize OpenVera verification intellectual property (VIP) to accelerate the development of complete verification suites. Till now, according to Catalyst Program's web site, 30 companies have joined the OpenVera Catalyst Program. The current available OpenVera VIP solutions are listed in Table 4.

The guidelines for the structured development of verification IP based on OpenVera can be found in Synopsys's white paper – Verification Intellectual Property (VIP) Modeling Architecture, Guide to Structured Development Using OpenVera, version 1.1. More information on OpenVera Catalyst Program and updated OpenVera verification IP solutions can be obtained at Catalyst Program's web site: www.open-vera.com/catalyst.

6 CO-SIMULATION IN CO-DESIGN

6.1 Introduction

Despite the fact that the correct functions of electronic systems, especially embedded systems, often require the hardware and software components closely co-operate together, the software and hardware development fields evolved along the separate paths through the end of the 20th century [102].

Traditionally, the split of the hardware and software development paths takes place in the early design cycle, usually on ad hoc basis [103]. After the partition between the hardware and software is decided, the software and hardware teams take different approaches and work independently with very little interaction. Later, during system integration, which usually happens after the hardware is fabricated and the physical prototypes are built, the software and the hardware are finally combined and tested together as a whole system. If problems were discovered at this stage, the software and/or hardware components had to be modified: the modification of the software is commonly used to work around the hardware inadequacies [104]; and the development of additional hardware may be necessary to compensate the software's poor performance [105].

Although in the past, many electronic systems were successfully built using the design approach described above, this conventional design approach is unable to manage today's design challenges - the short time-to-market, the enriched system functionality, the increased design density, the strict design constraints, etc.. The separation of hardware and software design paths made late bug discovery an inevitable scenario. Modifications on either hardware or software attempting to fix the bugs in the late design phases usually bring significant cost and elongated design cycle. In addition, with the traditional design approach, the exploration of the various design alternatives and the evaluation of the hardware-software tradeoffs are very restricted. Once the separate developments of the hardware and software proceed, the retraction of the hardware-software partition is extremely difficult and very expensive. Unfortunately, because different design options often lead to dramatically different cost-performance outcomes [106], it is always worthwhile to explore different design options and evaluate the hardware-software trade-offs before making a final design choice.

In order to address the problems associated with the traditional design approach and close the cost and the time-to-market gap caused by the differences in the hardware and software design flows [107], a unified and cooperative design approach, the hardware-software co-design, has been proposed and promoted. It is difficult to give co-design an accurate and comprehensive definition because different research groups tend to define it somewhat differently. For example, in [108], co-design is defined as the methodology, tools, and practices that support the integration of the hardware and software components during a system's design and development; and in [109], co-design is described as the approach that aims at providing an integrated environment for concurrent specification, validation, and synthesis of both hardware and software.

Notwithstanding the numerous definitions of co-design, Michaela Serra and William B. Gardner listed the major definitions of co-design in [110], which capture the essence of the area:

- the cooperative design of hardware and software components;
- the unification of currently separate hardware and software paths;
- the movement of functionality between hardware and software;
- and the meeting of system-level objectives by exploiting the synergism of hardware and software through their concurrent design.

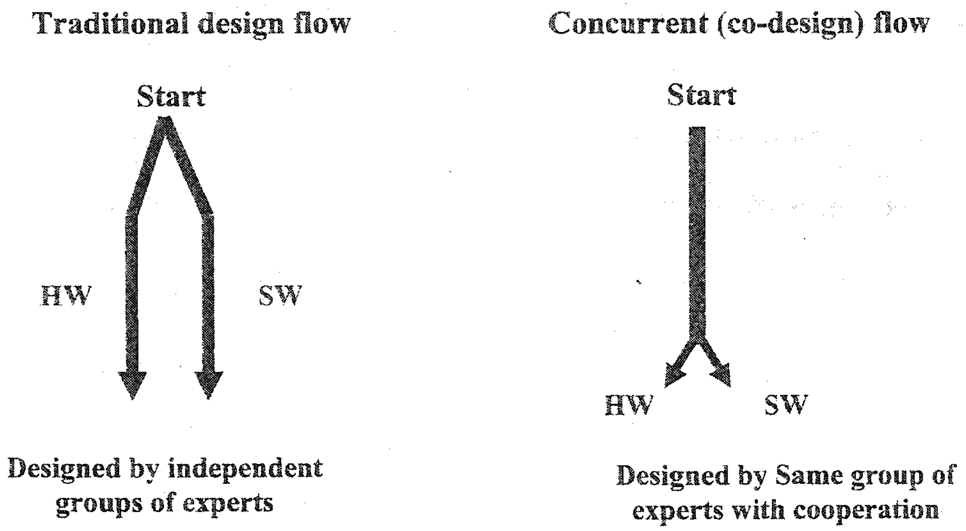

Traditional design flow
Start
HW
SW
Designed by independent groups of experts
Concurrent (co-design) flow
Start
HW
SW
Designed by Same group of experts with cooperation

Figure 10: Traditional design flow vs. co-design flow

Different from the conventional design approach, co-design focuses on a unified design environment. It emphasizes the use of the same integrated infrastructure for the development of both hardware and software to achieve the improved overall system performance, reliability, and cost effectiveness. In co-design, the unification of the traditionally separate hardware and software design paths cannot be characterized merely by the feedback sessions across the hardware and software teams or the weekly held designers meetings. The successful deployment of a co-design process dictates the use of an effective co-simulation platform and powerful tools that can support the exploration, prototype implementation, and rapid evaluation of the repartitioning of the functionality between hardware and software. Figure 10 (from [109]) shows a simple comparison between traditional design flow and the co-design flow.

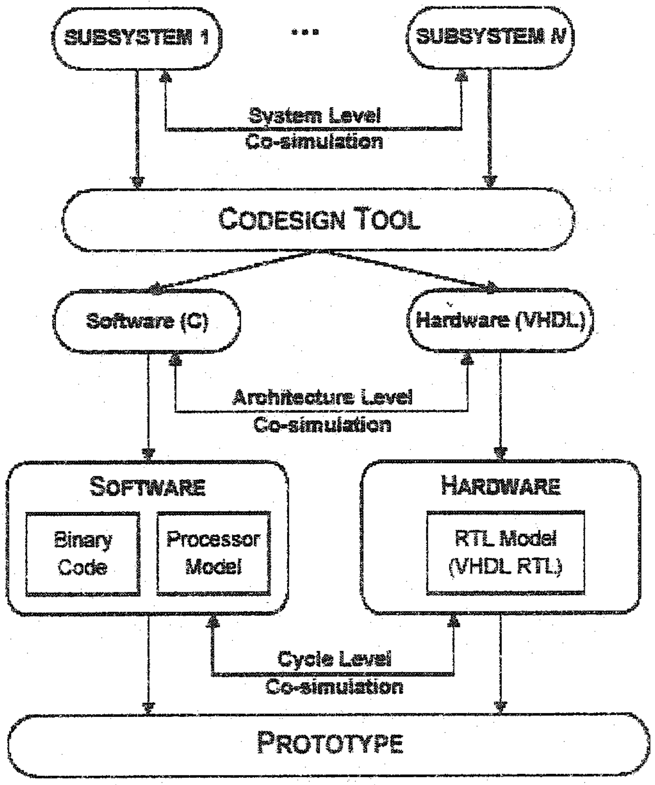


Figure 11: Co-design flow highlighting co-simulation at different abstraction levels.

During the past decade, co-design has gained enormous attention from both academia and industry. The early work on co-design came from academia [111], which can be exemplified by Ptolemy - a framework for hardware-software co-design and co-simulation developed at the University of California-Berkeley [112]. A few years later, commercial solutions, such as Mentor Graphics' *Seamless* [113], and the *EagleI* , *EagleV* developed by Eagle Design Automation (now merged with Synopsys), gradually surfaced in the market. Currently, research in the area of co-design concentrates on the following themes [110] [114]: *system modeling* - methodologies for specifying hardware/software systems; *system partitioning* - how to divide specified functions between hardware and software; *hardware-software co-synthesis* - the generation of the hardware and the software as well as the communication between them; and the *hardware-software co-simulation.*

Within a co-design process, co-simulation plays an indispensable role and is of crucial importance in the validation of the heterogeneous systems. In a simplistic way, co-simulation can be described as verifying if hardware and software function correctly together [115]. It usually combines the simulation of the software running on a programmable processor with the simulation of the weakly programmable (or fixed) hardware subsystems [109]. Like other hardware verification techniques, co-simulation aims at verifying the product's functionality as much as possible before the actual hardware is fabricated [116]. In the past, the separation of the hardware and software design paths severely confined the software developers' ability to test software components; and co-simulation was conducted only after the hardware is deemed to be working and stable [116]. With the maturity of the behavioral model simulation and the improvement of the simulation tools, co-simulation can now be adopted throughout a system's development cycle. Figure 11 (from [117] and [118]) presents different levels in a co-design flow that co-simulation can be performed.

With the adoption of co-simulation at different phases throughout the co-design process, co-simulation has become an imperative aid to many critical design tasks. For example, co-simulation is utilized in the tasks of exploring the hardware-software tradeoffs, evaluating the design alternatives, optimizing the hardware-software partitions, checking the correctness of the interface between hardware and software, assessing the hardware or software performance, and verifying the functions of the whole system. As co-design becomes a dominant design trend, tremendous research effort has been invested in the development of co-simulation environment to better serve the co-design needs.

This chapter surveys the co-simulation approaches and tools. Even though co-simulation is only one of the many essential steps in a co-design process, it is rather a rich field to explore, and because of this, the survey can merely serves as an introduction.

6.2 Co-simulation Approaches

Hardware-software co-simulation involves verifying both the hardware and software components of a system in a concurrent and interactive fashion. As a result of an endeavor of more than a decade in finding the effective and efficient methods of co-verifying both the hardware and software components of an electronic system, many commercial and non-commercial co-simulation tools and techniques are available. This section arranges the existing co-simulation solutions into groups and introduces them accordingly.

Depending on the classification criterion, co-simulation approaches can be categorized differently. Based on how the hardware and software components are glued together, co-simulation techniques can be grouped into two broad categories [115]: *techniques that require processor models* and *techniques that do not require processor models*. If the number of the needed simulators is considered, then co-simulation environment can be classified as the *homogeneous co-simulation* and the *heterogeneous co-simulation*. Moreover, if the geographical location of a co-simulation environment is taken into account, then the approaches can be divided as the *distributed co-simulation* and the *local co-simulation*.

6.2.1 Requiring Processor Models vs. Requiring no Processor Models

6.2.1.1 Processor Models

Not only can processor be modeled in both hardware and software, but also the software processor models can be built in various levels of abstraction with each abstraction level offering different accuracy/performance trade-off. Software models in the high level of abstraction offer good performance at the price of losing significant timing accuracy; software models with detailed timing information, on the other hand, often demand long simulation time.

6.2.1.1.1 Hardware Processor Models

When a processor is modeled in hardware, the co-simulation environment, as shown in Figure 12 (from [119]), often includes a board with the target processor or

FPGAs mounted on it [119]. The compiled software is loaded into the on-board memory and runs on the target processor during simulation. If the target processor is not available, then a gate-level description of the processor's functionality can be mapped onto the on-board FPGAs to emulate the target processor [120].

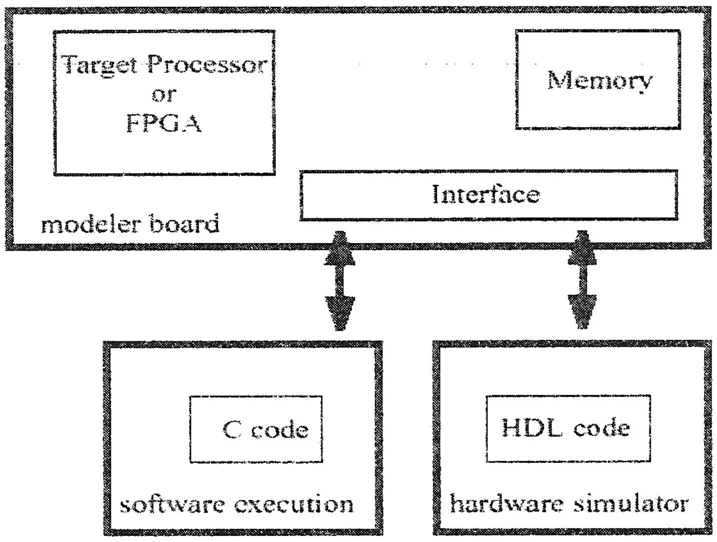


Figure 12: Hardware modeler

This scheme, in fact, is based on the same idea used by the hardware-accelerated simulation, which is introduced in section 2.2.4.1 of this survey. Modeling processor in hardware can increase simulation speed, which is important to the verification of the large gate-level designs. In [120], Dreike and McCoy stated that the speed of the entire simulation, using the hardware-accelerated environment, is controlled by either the software simulation or the communications overhead between the hardware processor and the simulator. Although the hardware modeled processor can provide improved simulation speed, to build up such a co-simulation environment and make the accelerators working correctly with gate-level models is costly and takes a lot of work [120].

Since the hardware-accelerated simulation is described earlier in the survey, co-simulation examples using hardware modeled processors will not be given here; but similar approaches can be found in section 3.3.

6.2.1.1.2 Software Processor Models

A Software processor model is a functional description of a processor [119], which can be built at different levels of abstraction. According to the performance/accuracy tradeoffs they offer, software processor models can be grouped into the nano-second accurate model, the cycle accurate model, the instruction set accurate model, and the bus functional model. A comparison among those software models can be found in Table 5.

- **The Nano-second Accurate Processor Model**

The nano-second accurate processor model provides the highest timing accuracy and the best debug capability among all the software models [115]. Because it captures the most detailed functional description of a processor, many internal transitions need to be calculated during simulation; therefore, this type of processor models suffers from low performance. According to James A. Rowson [115], the typical performance for this type of model is 1 to 100 instructions per second. Also, the nano-second accurate processor models are the hardest models to develop because they are often detailed to gate level. Consequently, when co-design is still at the early phase of verifying system specification, it would be inappropriate to use the nano-second accurate processor models for co-simulation.

- **The Cycle-accurate Processor Model**

The cycle-accurate processor model can be used to accurately simulate a processor's behavior at the cycle-accurate level. The software model of this type provides the correct transitions at each clock edge, but it does not ensure the exact delay time needed for a transition [119]. Because it includes less functional information of a processor comparing to the nano-second accurate model, the cycle accurate processor model offers the relatively improved simulation speed – 50 to 1000 instructions per second [115].

- **The Instruction Set Accurate Processor Model**

The instruction set accurate processor model is commonly included in the software development environments to assist software debugging process. It models the values of the internal registers and memory, and it can be used to accurately simulate the instruction fetch, decode, and execute units in a processor [121]. Usually, this kind of instruction emulation model provides the execution speed at the range from 2,000 to 20,000 instructions per second [115]; and the speed gain is obtained by ignoring the internal pipelines, hazards, and interlocks [115]. As a tradeoff to performance, the instruction set accurate processor models handle only approximate timing information or no timing at all [122].

- **The Bus Functional Model (BFM)**

The bus functional model of a processor is the least expensive and most readily available one, and it is often used to exercise and debug the hardware side of the hardware/software interface in a system [115][122]. The intent of the bus functional models is to model the bus transaction of a processor instead of the functionality of the processor [123]. It is normally derived based on a processor's bus specification document, not the functional processor description [124]. Hence, a BFM can only execute bus transactions on the processor bus, often with cycle accuracy, but cannot execute any instructions [121]. For this reason, a bus functional model is usually not considered as a processor model. The simulation performance of the bus functional model is limited by the hardware simulator used in co-simulation.

6.2.1.2 Techniques Require Processor Model

In the techniques that require processor models, co-simulation is performed by simulating the final machine code on a processor model [125]. During the course of this simulation process, the hardware and software components are linked together by the modeled processor. For that reason, the availability of the processor model becomes the premise of the possible application of this type of co-simulation technique.

Using processor models in co-simulation environment are fairly common [126][121]. Here one example is described to illustrate the rationale behind the choice of different processor models at different design stages.

Séméria and Ghosh, from Stanford and Synopsys Inc., reported their SystemC based design environment for hardware /software co-verification [121]. In this paper, Séméria and Ghosh also described, in their co-verification environment, how co-simulation can be carried out efficiently and effectively at various levels of abstraction and how the different processor models and techniques are used to speed-up co-simulations.

Figure 13 shows an un-timed co-simulation model that is used in the early design stages where the hardware-software co-simulation first comes into the proposed methodology. At this stage, the system's architectural specification is created, and the objective of co-simulation is to validate the architecture and determine its performance. The simulation speed needs to be extremely fast at this level because various architectures may need to be explored via simulation; and models used in simulation at this level tend to be more abstract in order to satisfy the high performance demand. Therefore, for the processor, only a bus functional model is used. This co-simulation model is un-timed because the software runs on the host processor and its execution time is not accurate.

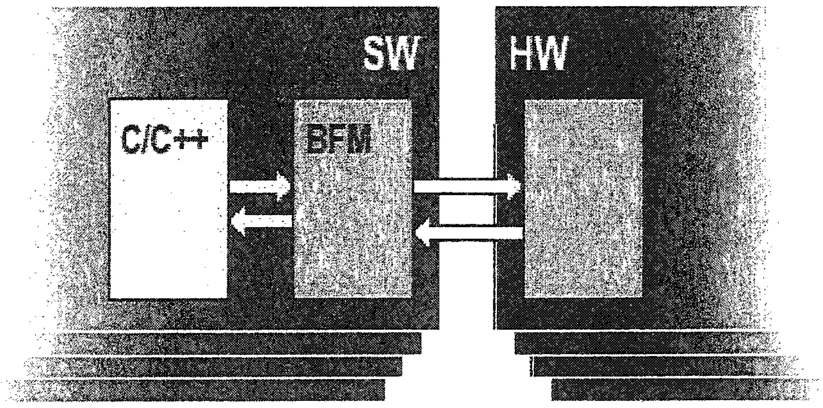


Figure 13: Séméria and Ghosh's untimed co-simulation model

Once the architecture choice is made, the hardware and software teams work in parallel to refine the individual hardware and software blocks by adding the necessary implementation details and specifying the constraints for synthesis. In the course of those refinement processes, co-simulation must be conducted to ensure that the system, as a whole, still works. At this point, simulation speed slows down as the refined hardware and software blocks are much more detailed. Hence, a bus functional model alone can no longer fulfill the co-simulation task, and an instruction set accurate processor model (ISS) is employed in conjunction with the BFM. The ISS executes the instruction for the target processor, and the BFM handles the communication among the software and hardware components of the simulated system. Figure 14 depicts the co-simulation model used at this level.

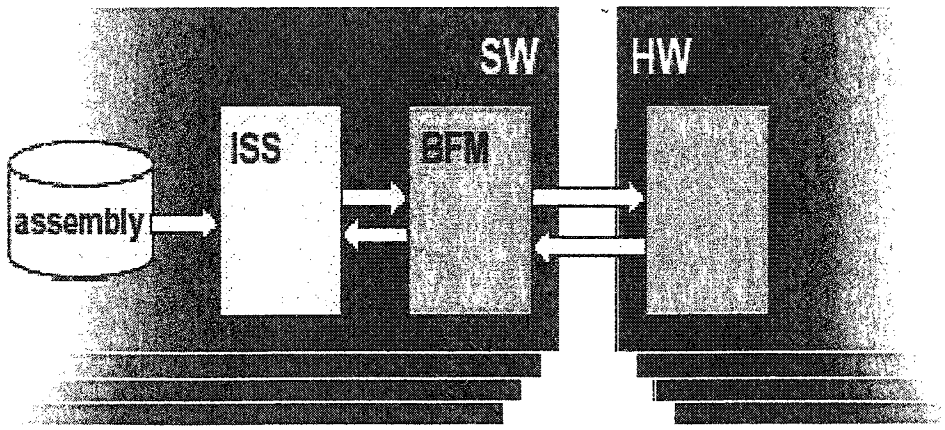


Figure 14: Séméria and Ghosh's cycle-accurate co-simulation model

The bus functional processor model, used in both co-simulation schemes (Figure 13 and Figure14), is derived from a SystemC class called *sc_module* using C++

inheritance. Programming interface to the C/C++ software and to the instruction set processor model is provided as the member functions of the BFM class; the functionality of the BFM itself is modeled as a set of finite-state machines that can execute in parallel. Notably, the BFM constructed in the SystemC environment has a relatively fixed programming interface, which allows the easy swap of one processor model to another without changing the C/C++ source code. This feature works ideally when different architectures with different processors are explored.

The instruction set accurate processor model reads the assembly code written for the given architecture and simulates it on a host machine. In terms of choosing the instruction set accurate processor model, the proposed co-verification approach offers flexibility to the designers. For example, if the purpose of co-simulation is to verify the functional correctness of an application written in assembly code, then a fast ISS, which translates the instructions of the target architecture into the instructions of the host machine, can be used; if the timing and interfaces between the different components of the system need to be verified, an ISS can be used in conjunction with a BFM to provide accurate timing.

After synthesis, the gate-level co-simulation needs to be performed to verify the final system implementation, the result of system integration, and whether or not the system satisfies the constraints specified in the initial system specification. At this point, Séméria and Ghosh suggested that any co-simulation techniques are in use today be employed.

6.2.1.3 Techniques Require No Processor Model

For the techniques that do not use processor models, co-simulation is usually performed by compiling the software for a computer and linking the executable to a bus functional model of the processor, which is simulated in conjunction with the hardware component [125]. In such a case, mechanisms, such as synchronizing handshakes, virtual operating systems, etc., are utilized to handle communications among the hardware and software components [115]. An example of co-simulation technique that requires no processor models is described below.

Operating systems, such as UNIX, usually provide the facility for running processes to communicate with each other. In co-simulation, this inter-process communication facility can be used to link the hardware and software components. In [127], Becker et al. presented an engineering environment that links the software components of a system to the simulation of the hardware components. Employing this environment, they successfully performed co-simulation of a network interface unit (NIU) on a distributed network using Cadence's *Verilog-XL* simulator and the UNIX sockets. The software and hardware components were modeled in C++ and Verilog respectively. In the co-simulation environment, the software components were implemented as separate programs, which interact with the hardware simulation via UNIX interprocess communication (IPC) mechanisms. To accomplish the hardware and software communication, both interface functions in the software components and the simulation modules of the hardware components were modified. The major drawback of Becker's technique is that it does not accurately simulate the relative speeds of the hardware and software components; in other words, this solution does not include timing evaluation.

In the co-simulation techniques that do not use processor models, the software directly runs on the host processor; and thus the nearly real time execution of software can be achieved [111]. Unfortunately, as illustrated by Becker's engineering environment, the gain of the real time execution of software is under the loss of timing accuracy. There are attempts to decrease this disadvantage by back annotation of the software runtime [122] [128]. For instance, in Bassam et al.'s co-simulation technique [122], software is modeled by using behavioral VHDL constructs and annotated with timing information that is derived from basic block-level timing estimates. For each block of the software, the delay information was derived through using timing estimation methods introduced in [129] [130]; and the derived software timing estimation was further utilized by a process emulating the target RTOS behavior to synchronize the processes modeling the software tasks with those modeling the hardware components.

6.2.1.4 The Choice of Co-simulation Techniques

The choice of different processor models could have direct impact on co-simulation performance, timing accuracy, and the visibility of internal state for debugging purpose [115]. The optimum selection of processor models should always take into account the specific co-simulation goal dictated by the corresponding development phase. For example, at system level (see Figure 11), the goal of co-simulation is to characterize the system's functionality, thus the use of nano-second accurate processor model is of no necessity.

In [115], Rowson pointed out that the processor model availability dominates the choice of the co-simulation technique. Rowson also did a comparison of hardware-software co-simulation techniques based on the tradeoffs among a number of factors, such as performance, timing accuracy, model availability, and visibility of internal state for debugging. Partial result of Rowson's comparison is illustrated in Table 5.

Table 7: Rowson's comparison of hardware/software co-simulation techniques

	speed (instructions. per second)	debugging ability	processor model requirements
hardware modeller	10-50	no processor state	timing only
logic emulation	fast	limited	none
nano-second accu-rate	1-100	excellent	hardest
cycle accurate	50-1000	good	hard
instruction set	2000-20,000	OK	medium
synchronized hand-shake	limited by hardware simulator	no processor state	none
virtual operating system	fast	no processor (and no hardware) state	easier
bus functional	limited by hardware simulator	no processor state	easier

6.2.2 Homogeneous Co-simulation vs. Heterogeneous Co-simulation

Based on the number of simulators used, co-simulation approaches can be grouped into the *homogeneous co-simulation* and the *heterogeneous co-simulation*. In the *homogeneous co-simulation*, only one simulator is required; and in the *heterogeneous co-*

simulation, the number of simulators employed is determined by the number of different languages or models of computation (MOC) used in modeling the system.

6.2.2.1 Homogeneous Co-simulation

The use of single language and a uniform modeling paradigm is fundamental to the adoption of the homogeneous co-simulation. In other words, modeling the whole system in one language made it feasible to utilize only one simulator for simulation of both the hardware and software components. The major advantage of this approach is that it eliminates the communication overhead among different simulators. In addition, describing a system in one language allows the designer to move functionality from hardware to software and vice versa easily, which eases the exploration of different architectures and hardware-software partitions.

In Séméria and Ghosh's co-verification scheme [121] introduced in section 4.2.1.2 of this survey, both the hardware and software components throughout the design flow are described in C/C++. There are no overheads associated with interfacing HDL simulators with the software world; and only an instruction set simulator along with a bus functional model (BFM) is used for co-simulation.

In [122], Bassam et al. described a co-simulation technique in which both the hardware and software components were modeled in VHDL. This technique does not require the use of inter-process communication, nor a C language interface for the software components; any commercial VHDL simulator can be employed in this co-simulation method. The proposed co-simulation methodology is heavily based on the use of the software and hardware synthesis, and POLIS co-design environment for reactive embedded system, which is introduced in section 4.3.2 of this survey, was employed to synthesize the software and hardware components. In order to achieve fast simulation speed, the reported co-simulation technique ignores some aspects of the final embedded system implementation. For instance, it ignores the overhead due to the scheduling mechanism and the cost of inter-processor or hardware/software communication.

6.2.2.2 Heterogeneous Co-simulation

When different languages or models of computation (MOC) are used to describe a system, a dedicated simulator is needed for each language or MOC employed. The major challenge of this kind of co-simulation approach is the construction of an efficient bridge among the heterogeneous simulators.

In [117], Amory et al. presented a heterogeneous and distributed co-simulation environment in which the communication among simulators is carried out using a co-simulation backplane. Figure 15 shows the general structure of their proposed co-simulation environment.

The software modules are written in C and simulated using *gcc*, and the hardware modules are described in VHDL and simulated by *QuickHDL*. The integration of simulators, here *gcc* and *QuickHDL*, to the co-simulation backplane is through UNIX sockets. The communication library, *ComLibC and ComLibVHDL,* has three functions - initialization (*csiInitialize()*), send data (*csiSend()*), and receive data (*cisReceive()*) - that are utilized by simulators to carryout socket communication with the co-simulation backplane. A *coordination file* contains the information of each module: its unique name, the language in which the module is described, the simulator used to validate it, the CPU where the module will be simulated, and the name and direction of each external module pin. The inter-module connections are also specified in the *coordination file.* During simulation, the co-simulation backplane builds its internal data structure based on the information provided by the *coordination file* to control the simulation and enable the routing of messages among modules. The proposed co-simulation environment does not support cycle level co-simulation because the backplane does not have a global synchronization mechanism; however, the backplane offers great flexibility in terms of integration of new languages because it is independent from the simulators.

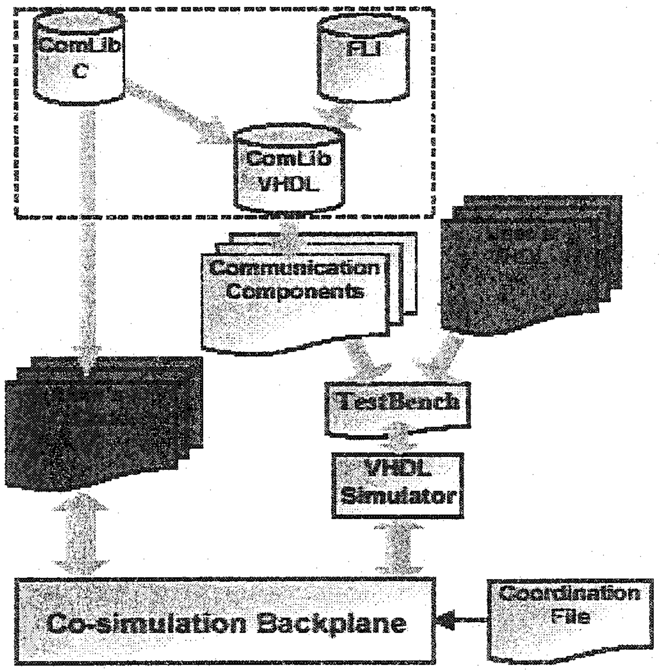


Figure 15: General structure of Amory et al.'s co-simulation environment

6.2.3 Geographically Distributed Co-simulation vs. Local Co-simulation

The evolvement of the *geographically distributed co-simulation* is closely in connection with the intellectual property (IP)-based design using the Internet [131] [132] or the globally distributed design [133] [134]. Such a co-simulation environment allows parallel execution of simulators in geographically distributed machines over a local area network (LAN) or a wide area network (WAN) [117]. Conversely, in the *local co-simulation* environment, the simulators run on machines located within a confined local environment; and no network is used for message exchange among simulators or modules during simulation.

6.2.3.1 Co-simulation Environment Structure

Since more than one simulator is employed, the *geographically distributed co-simulation* can be treated as a special case of the *heterogeneous co-simulation*, which is introduced in section *6.2.2.2*. Consequently, the general structure for the *heterogeneous co-simulation* is often suitable for the *geographically distributed co-simulation*. For instance, Amory et al.'s structure for a heterogeneous co-simulation environment (cross reference 4.2.2.2) also supports geographically distributed co-simulation [117].

In [135], BA de Mello and FR Wagner presented a generic architecture to support environments for geographically distributed co-simulation, called Distributed Co-simulation Backbone (DCB). The proposed co-simulation backbone is based on HLA (High Level Architecture), which was originally developed by the US Department of Defense as a standard for military simulation interoperability within the US; and in the year of 2000 HLA was adopted as a non-military standard by the IEEE [136].

HLA offers a common architecture for the cooperative and distributed execution of individual simulations [135]. In this architecture, several simulation systems (represented by simulators), called "federates," are combined together into one big simulation, called the "federation." HLA is specified by three main parts [137]: (1) "A set of rules which must be followed to achieve proper interaction of simulations in a federation. These rules describe the responsibilities of simulations and of the runtime infrastructure (RTI) in HLA federations." (2). "Definitions of the interface functions between the runtime infrastructure and the simulations participating in a HLA federation." (3) "The prescribed common method for documenting the information contained in the required HLA Object Model for each federation and simulation." A detailed description of the IEEE 1516 standard for HLA can be found at the web site: http://www.ieee.org.

Mello and Wagner's co-simulation backbone (DCB) [135] is constructed based on the HLA standard, and it aims at offering a generic mechanism for communication and cooperation services among the distributed heterogeneous federates (simulators). Figure 16 (from [135]) depicts the architecture of Mello and Wagner's DCB.

In this scheme, *the ambassador's paradigm* [138] is used to provide services for the cooperation among simulators (federates); and gateways [138], implemented as part of the ambassadors, translate data formats according to the destination of the data sent through the DCB. The distributed co-simulation backbone infrastructure, shown in Figure 16, is general-purpose, which means that the integration of new elements (simulators) into a federation will not affect the DCB. However, when a new simulator is integrated into a co-simulation environment, two ambassadors must be developed - one for simulator and one for DCB. As part of the DCB development work, the authors also built a supporting environment for the DCB, which offers services and resources for the semi-automatic generation of the ambassadors.

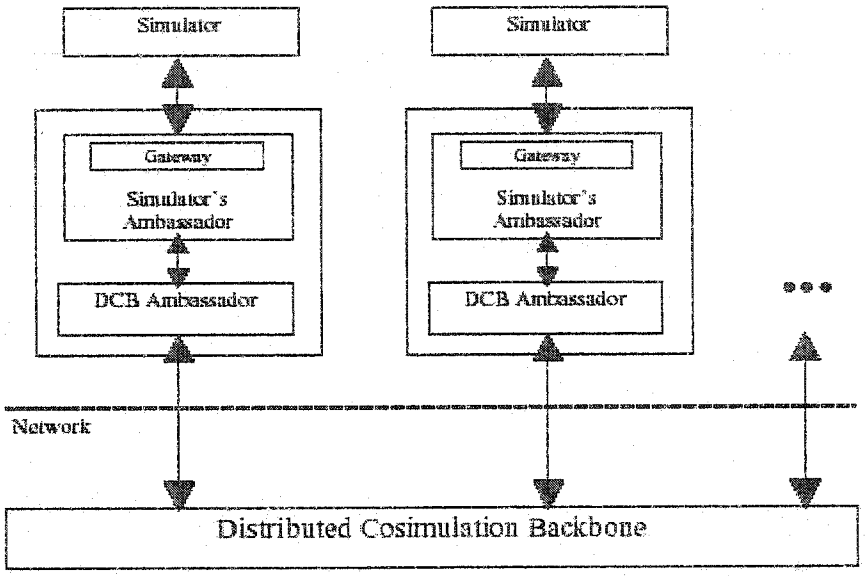


Figure 16: Architecture of the DCB

6.2.3.2 The Benefits of Geographically Distributed Co-Simulation

The geographically distributed co-simulation approach offers some benefits that do not present in the local co-simulation [117] [139]:

- It decentralizes the project by allowing design and validation of a system under development by geographically distributed teams;
- It allows designers to simulate a system that consists of remotely located IP blocks without requiring local copies of the IP description (source code); hence, IP providers and EDA vendors can let their IP blocks and proprietary tools, such as high-performance hardware emulators, be accessed remotely while protecting their IP rights and tool licenses;

- It provides a fundamental infrastructure for resource sharing through remote tool access and IP simulation.

6.2.3.3 Performance Issue in Geographically Distributed Co-Simulation

In geographically distributed co-simulation, the transfer of messages between simulators can cause considerable network communication overhead. As a result, geographically distributed co-simulation faces a significant problem in terms of co-simulation performance.

There are two kinds of messages transferred among simulators in the geographically distributed co-simulation: (1) event-carrying messages and (2) null messages that are used for simulator synchronization only [139]. The performance optimization methods of geographically distributed co-simulation lie in the reduction of both the event-carrying messages and null messages [139].

Yoo et al. in [139] proposed a technique for performance improvement of geographically distributed co-simulation, which is based on a new concept called *hierarchically grouped message* (HGM). This HGM concept utilizes the fact that transmitting one large message is faster than transmitting multiple small-sized messages one by one; and, likewise, the network communication overhead does not strictly depend on the size of the messages being transferred, instead, it strongly relates to the number of physical messages transferred. Based on the HGM concept, Yoo et al.'s method hierarchically groups messages transferred between simulators in a short period of time into a single physical message to reduce the number of physical messages needed to be transferred during co-simulation, and thus improves the performance.

More approaches to improving the performance of geographically distributed co-simulation can be found in [140] [141] [142] [143].

6.3 Co-simulation Tools

While hardware/software co-design is rapidly becoming an essential capability for electronic systems with mixed hardware and software components, EDA vendors along with research institutions are starting to launch programs to create co-design systems. As a result, more and more commercial tools and research products become available to the co-design community. Since co-simulation is a widely adopted co-verification method in hardware-software co-design, co-simulation tools are often developed as an important part of the co-design systems.

The rest of the section contains two parts: the first part gives relatively detailed descriptions of four co-simulation tools that have made substantial impact on the advancement of co-simulation techniques; and the second part collects the information of various commercial and non-commercial co-simulation tools.

6.3.1 Ptolemy

Ptolemy [112] [144] [145] [146] is a heterogeneous simulation and design environment that supports multiple models of computation. It was developed as part of a design project – Ptolemy -conducted in the EECS department at the University of California, Berkeley since 1990.

The Ptolemy offers two possible execution styles: one with Ptolemy interactive graphic interface and another without the graphic interface. As depicted in Figure 17(a), when using the Ptolemy interactive graphic interface, two Unix ™ processes need to be started. The first process contains the user interface (VEM) and the design database (OCT) and the second process contains the Ptolemy kernel. If Ptolemy runs as a single process without the graphical user interface, as shown in Figure 17(b), the textural interpreter based on the Tool Command Language, TCL, can be used to provide a textual interface for the user.

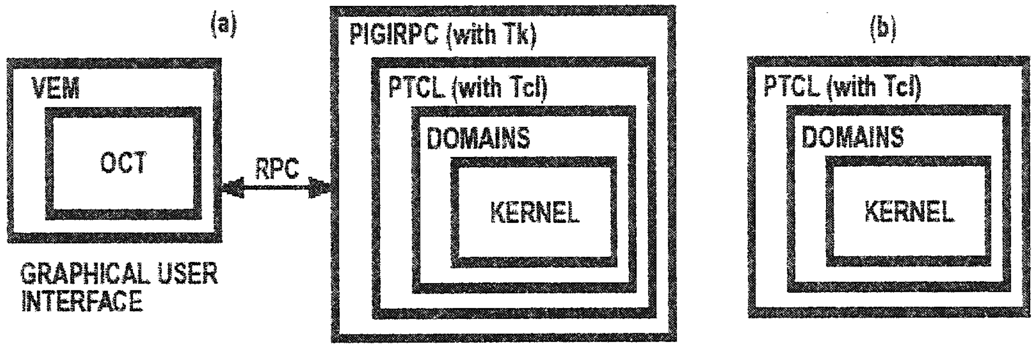


Figure 17: The overall organization of Ptolemy version 0.7

Ptolemy is written in C++. It uses an object-oriented software technology, such as polymorphism and information hiding, to model each subsystem and integrate these subsystems into a whole. A family of C++ class definitions forms the software infrastructure, called *Ptolemy kernel*, upon which the specialized design environments, called *domains*, can be defined by creating new C++ classes derived from the basic classes in the *Ptolemy Kernel*. In Ptolemy, the simulation of the heterogeneous systems described in multiple design styles is allowed; and the *domain* is used to realize a computational model appropriate for a particular type of subsystem. For example, the

Thor domain is defined in Ptolemy for the RTL hardware simulation. In Table 6 (from [147]), some simulation domains implemented in Ptolemy are listed, and the updated list along with descriptions can be found in [146].

Table 8: Simulation domains implemented in Ptolemy

Name	Expansion	Principal Use
SDF	synchronous dataflow	synchronous signal processing
DDF	dynamic dataflow	asynchronous signal processing
BDF	boolean dataflow	asynchronous signal processing
MDSDF	multidimensional dataflow	multidimensional signal processing
DE	discrete event	communication network modeling and determinate high-level system modeling
FSM	finite state machines	control
HOF	higher-order functions	graphical programming
Thor	(name given at Stanford)	RTL hardware simulation
MQ	message queue	telecommunications switching software
PN	process networks	real-time systems
CP	communicating processes	communication network modeling nondeterminate system modeling

Ptolemy enables designers to create their own co-simulation environments for the validation of heterogeneous systems [119]. The obtained co-simulation environment comprises several domains that define different simulation behavior; and Ptolemy pre-determines the basic internal structures, such as Blocks, Targets, and associated Schedulers, for domains and ensures the proper communications among them.

Ptolemy is a non-commercial product and can be downloaded from Ptolemy project's web site: http://ptolemy.eecs.berkeley.edu. Ptolemy version 0.7.1, released on June 12, 1998, is the stable production release of Ptolemy Classic; and Ptolemy 0.7.2devel is also available for testing to experienced Ptolemy developers. Although the Ptolemy groups' work has shifted to a new Java-based environment called Ptolemy II, there remains an extensive network of active users. Particularly, many co-design and co-simulation tools are built on Ptolemy. For example, Pia, a co-simulation tool developed at the University of Washington in Seattle is based on Ptolemy domain; and PeaCE, a co-design environment for rapid development of heterogeneous digital systems, uses Ptolemy extensions. More information and the latest updates on Ptolemy project can be found at the web site: http:// www.ptolemy.eecs.berkeley.edu.

6.3.2 POLIS

POLIS [148] [149], a hardware-software co-design framework for reactive embedded systems, has been developed at UC-Berkeley. The computation model employed in POLIS is a single finite state machine-like representation, known as *Co-design Finite State Machine* (CFSM). Each element of a network of CFSMs describes a component of the system to be modeled, and both the hardware and software components perform the same computation for each CFSM transition. However, the hardware and software elements have different delay characteristics: a synchronous hardware implementation of CFSM can execute a transition in one clock cycle, but a software implementation will probably require more than one clock cycle. Ultimately, the elements in the CFSMs that specify a system will be synthesized in hardware or software.

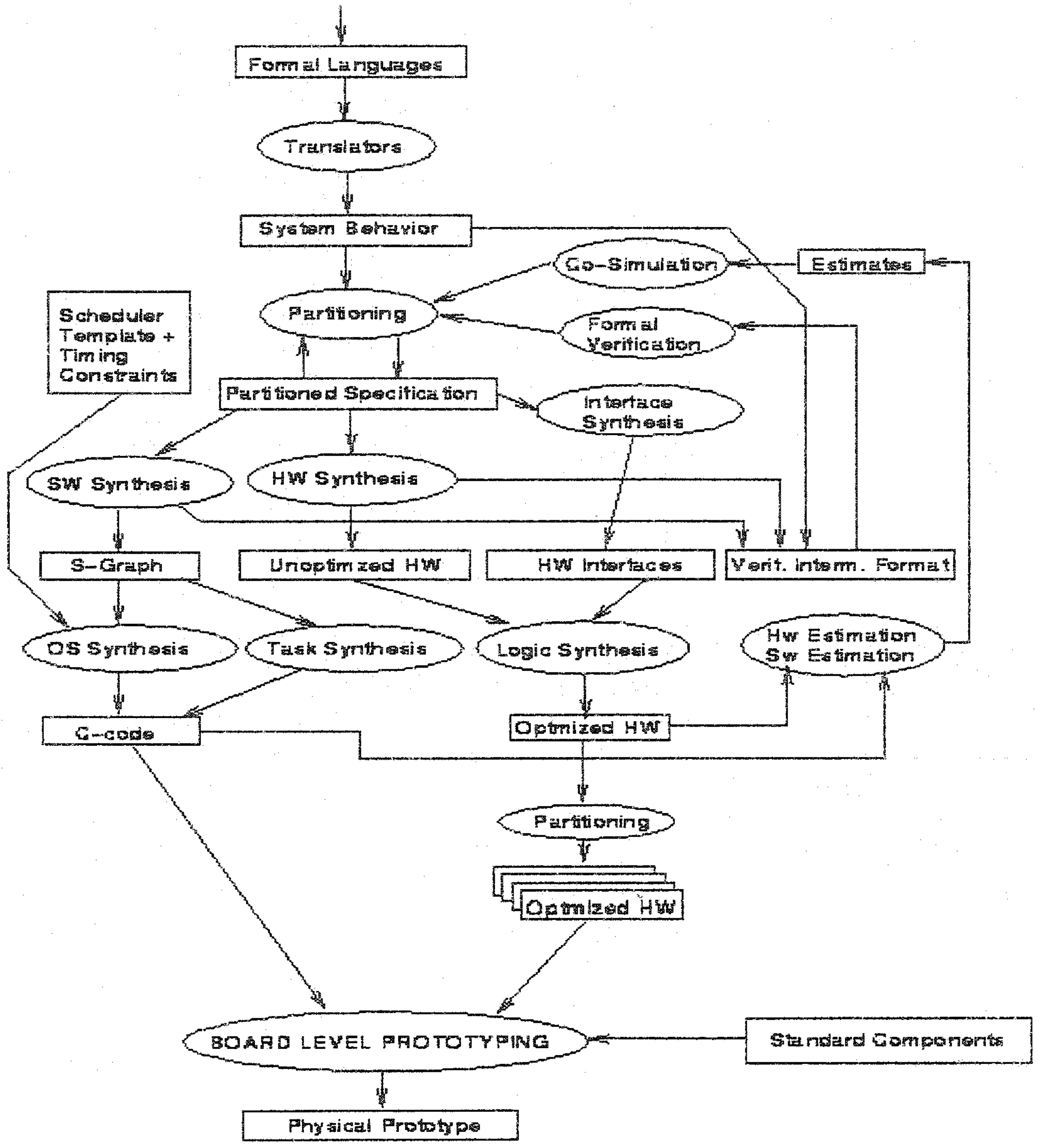


Figure 18: Design flow implemented in the POLIS system

POLIS can directly translate system specifications written in a high level language, like ESTEREL, graphical FSMs, and subsets of Verilog or VHDL, into CFSMs. For co-simulation, POLIS currently utilizes Ptolemy as its simulation engine; but it is not limited to Ptolemy. Since VHDL code along with all the co-simulation information is also an output of POLIS, any commercial VHDL simulator can be used for co-simulation purpose in POLIS. For instance, the Bassam et al.'s co-simulation technique introduced before is based on POLIS co-design environment in which a commercial VHDL simulator is used to perform co-simulation [105]. During co-simulation, POLIS allows designers to dynamically choose the different hardware or software implementation for each CFSM, the type and clock speed of the processor on which the software is running, and the type of scheduler.

Figure 18 depicts the design flow that is currently implemented in the POLIS system, and a detailed explanation of the design flow can be found at the web site: http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html. POLIS can be obtained from the web site: http://www-cad.eecs.berkeley.edu/~polis/; and questions about POLIS can be asked via email: polis-questions@ic.eecs.berkeley.edu.

6.3.3 Seamless CVE

The *Seamless Co-Verification Environment* [150] [148], a product from Mentor Graphics Corporation (www.mentor.com), first came to market in 1996. It was developed to address the growing need of co-verifying both the hardware and software components of embedded systems, especially prior to fabricating ASIC and building a hardware prototype.

Seamless supports simulation of the entire system and provides system-wide debug features. In the Seamless Co-Verification Environment, the connections among the hardware and software simulators are established based on a tool-independent backplane via open application programming interfaces (APIs) [119]. This connection establishment means enables Seamless to incorporate a wide range of hardware and software simulators. For example, it supports ModelSim (VHDL & Verilog), Verilog-XL, VCS, IKOS Voyager, etc. for hardware simulation.

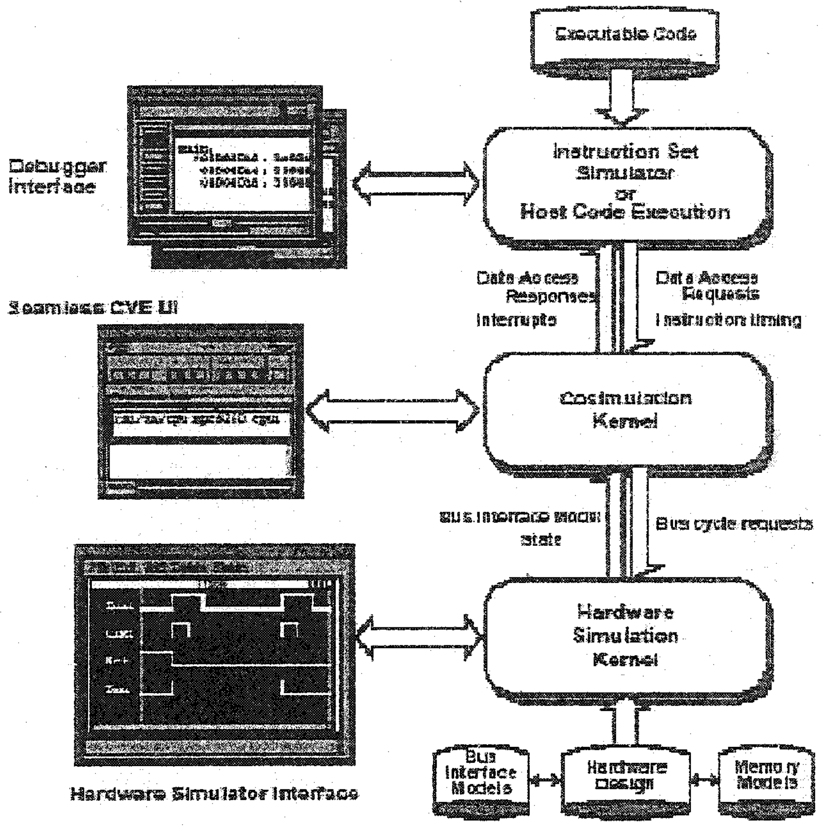
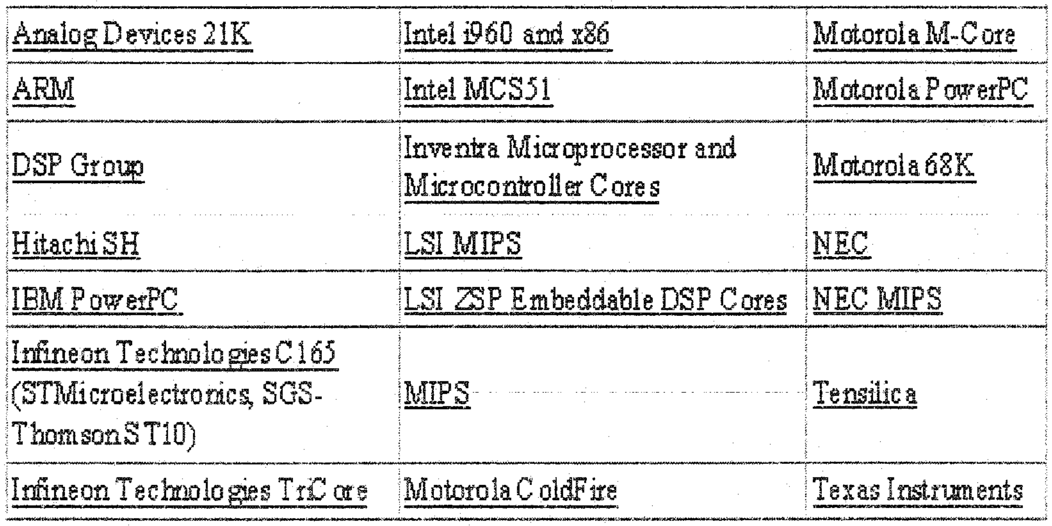

Executable Code
Debugger Interface
Instruction Set Simulator or Host Code Execution
Data Access Responses Interrupts
Data Access Requests Instruction timing
Seamless CVE UI
Cosimulation Kernel
Bus interface model state
Bus cycle requests
Hardware Simulation Kernel
Hardware Simulator Interface
Bus Interface Models
Hardware Design
Memory Models

Figure 19: Seamless CVE architecture

Seamless CVE has three major components: (1) the instruction set simulator (ISS), (2) the co-simulation kernel, and (3) the hardware simulator interface and hardware simulation kernel. Figure 19 (from [152]) depicts the architecture of *Seamless CVE*.

The instruction set simulator (ISS) performs the software portion of a co-simulation session. It fetches, decodes, and executes instructions; it reads and writes memory and I/O data; and it simulates the processor's registers and other internal data handling. *The co-simulation kernel* acts like a middle agent, it controls the communication between the software simulator and the hardware simulator during a co-simulation session. For example, it receives all address-space access requests from the instruction-set simulator (ISS), determines whether to pass those requests to the bus interface model or service the requests directly through local memory without hardware bus cycles, and reports the number of clock cycles for executing an instruction by ISS to the hardware simulator. *The hardware simulator interface and hardware simulation kernel* handles the hardware portion of a co-simulation session. It receives request for bus cycles and starts the necessary logical operations to make pin transitions.

Table 9: Seamless CVE processor families



Analog Devices 21K	Intel i960 and x86	Motorola M-Core
ARM	Intel MCS51	Motorola PowerPC
DSP Group	Inventra Microprocessor and Microcontroller Cores	Motorola 68K
Hitachi SH	LSI MIPS	NEC
IBM PowerPC	LSI ZSP Embeddable DSP Cores	NEC MIPS
Infineon Technologies C165 (STMicroelectronics, SGS-Thomson ST10)	MIPS	Tensilica
Infineon Technologies TriCore	Motorola ColdFire	Texas Instruments

Seamless is supported by a wide range of Processor Support Packages (PSP), and each PSP includes the XRAY (software simulator) source-level debugger, an instruction set simulator, and a bus interface model. Table 7 (from [153]) summarizes current PSP processor families, and the updated information on PSP can be found at the web site: http://www.mentor.com/seamless/psp_listings.html.

As part of the Seamless Version 4.3 released in April 2002, Mentor Graphics extended its Seamless co-verification environment's capability to include the C-Bridge technology [154]. The C-Bridge incorporates C and C++ hardware descriptions, testbenches, and protocol models into Seamless co-verification sessions to promote high-level modeling. Latest information on C-Bridge technology can be obtained from www.mentor.com/seamless. Also, a free Seamless CVE informational CD, which includes an introduction to Seamless and an overview presentation of Seamless, can be obtained from the web site: http://www.mentor.com/seamless/cd/cd_reques.cfm.

6.3.4 Eaglei

Eaglei, from Synopsys Inc., is a high performance Hardware/Software co-design and co-verification tool. It allows users to model portions of a to-be-built ASIC or PC board with HDL simulation, while running software debugging tools on a fast processor model [155]. Eaglei extends its virtual prototyping capabilities by linking up with industry leading RTOS simulators, and these extended virtual prototyping capabilities allow software developers to integrate their application code and operating system together with the virtual hardware prototype before the target hardware is available. Hence, hardware drivers can be fully tested before the hardware is built.

Unfortunately, Synopsys discontinued its Eaglei product about two years ago [156], and not much information on Eaglei can be found, even from Synopsys' web site.

6.3.5 Other Co-simulation Tools

In Table 8, a collection of tools that support hardware-software co-simulation are listed. More detailed information on each included co-simulation tool can be found at the corresponding web site under the column "Comment/Information."

Table 10: Co-simulation tools

Co-simulation Tool	**Comments / Information**
CoSim	TIMA Lab at Institute National Polytechnique, Grenoble
COSYMA	Braunschweig, http://www.ida.ing.tu-bs.de/home.e.html
CoCentric System Studio	Synopsys Inc., http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html
CoWare N2C® Products	CoWare Inc, http://www.coware.com
Eaglei	Synopsys Inc. This product is discontinued
PeaCE	National Soul University of Korea, http://peace.snu.ac.kr/research/peace/
Pia	University of Washington, http://www.cs.washington.edu/homes/hineskj/PiaMain.html
POLIS	Berkeley, http://www.cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html
Poseidon	Stanford
Ptolemy	Berkeley, http://ptolemy.eecs.berkeley.edu/
Seamless CVE	Mentor Graphics Corp.

SUCCESS™	Dolphin Integration, http://www.dolphin.fr/medal/success/success_overview.html
VCC	Candence Design Systems Inc, http://www.cadence.com/products/incisive.html
VCI	TIMA, http://tima-cmp.imag.fr/Homepages/valderr/Vci/vci.html
Virtual-ICE	Yokogawa Electric

7 CONCLUSION

This thesis presents a survey on hardware design verification. Generally speaking, there exist two broad approaches to hardware design verification: simulation-based verification and formal verification. The thesis contains information on both approaches, but with an emphasis on simulation-based verification. The survey covers four important aspects of simulation-based verification: simulation acceleration, simulation vector generation, verification environment construction, and hardware-software co-simulation.

In the last several years, formal verification has been progressively gaining acceptance in the hardware development industry, but design verification of today's digital systems still relies mainly on simulation. Due to their slow speed, simulation-based approaches lose their effectiveness rapidly when design sizes exceed half million gates. In order to speedup the simulation process, traditional interpreted simulators are replaced by compiled simulators; event-driven simulators lost their popularity to cycle-based simulators; server farms consisting of a cluster of networked computers are utilized for computationally intensive simulation tasks; and simulation-emulation co-operation techniques and tools are widely adopted to boost simulation efficiency.

The generation of simulation vectors is of great importance because the quality of simulation vectors often has direct impact on simulation performance. Since it is infeasible to exhaustively simulate a design, traditionally, directed testing and random testing are widely adopted by industry. In directed testing, simulation vectors are manually created; and in random testing, simulation vectors are randomly generated. During the last few years, new ideas of creating simulation vectors start to develop in both academia and industry. Based on the similarity between hardware design verification and physical fault testing, methods of borrowing test sets from physical fault testing to detect design errors are proposed. More notably, a large amount of research work on simulation vector generation via semi-formal methods has been reported. These studies attempt to make the simulation-based verification more efficient by using coverage metrics as heuristic measures to guide the generation of the simulation vectors, and to quantify the verification completeness.

The construction of verification environment is a fundamental step in simulation-based design verification, and it is also a process that consumes a considerable amount of time and resources. In order to improve verification productivity, EDA industry as well as many system design companies has invested significant amount of effort in developing new methodologies and tools to accelerate the construction of verification environment. As a result, new languages targeting at hardware verification were developed and some of them were already turned into open source and went on the road towards standardization; layered testbench architectures were proposed and adopted; and EDA vendors began to offer testbench automation tools; furthermore, the concept of intellectual property (IP) and the use of verification intellectual properties (VIP) are getting more and more popular in the hardware verification community.

Hardware and software design should be closely coupled, but they remain largely in separate worlds in today's design environment. As co-design gradually becomes a design trend, co-simulation, which is an essential part in the process of co-design, has also gained tremendous attention. There are many ways of classifying co-simulation techniques: based on how the hardware and software components are glued together, co-simulation techniques can be classified as *techniques that require processor models* and *techniques that do not require processor models*; based on the number of simulators needed, the co-simulation environment can be classified as *homogeneous co-simulation* and *heterogeneous co-simulation*; moreover, based on the geographical location of a co-simulation environment, the approaches can be categorized as *distributed co-simulation* and *local co-simulation*. Co-simulation tools have been developed by both EDA industry and research institutions. Among the commercially available co-simulation tools, Mentor Graphics' *SEAMLESS* is the most popular one.

LIST OF ACRONYMS

API	Application Program Interface
ASIC	Application Specific Integrated Circuit
ATPG	Automatic Test Pattern Generation
BFM	Bus Functional Model
CFSM	Co-design Finite State Machine
DCB	Distributed Co-simulation Backbone
DMA	Direct Memory Access
DUT	Design Under Test
EDA	Electronic Design Automation
FSM	Finite State Machine
ESS	Extended Synthesizable Subset (in Superlog)
FPGA	Field-programmable Gate Array
HDL	Hardware Description Language
HDVl	Hardware Design and Verification Language
HGM	Hierarchically Grouped Message
HLA	High Level Architecture (for Simulation)
HVL	Hardware Verification Language
HW	Hardware
IP	Intellectual Property
IPC	Interprocess Communication (in UNIX)
ISS	Instruction Set Simulator
MOS	Model Of Computation

OSCI	Open SystemC Initiative
PCI	Peripheral Component Interconnection
PD	Probability Distribution
PLI	Programming Language Interface (in Verilog)
PSM	Program State Machine
PSP	Processor Support Packages (in SEAMLESS)
RTL	Register Transfer Level
RTOS	Real Time Operating System
SCV	SystemC Verification Library
SSL	Single Stuck-at Line
STOC	SpecC Technology Open Consortium
SW	Software
UTS	Universal Test Set
VIP	Verification Intellectual Property
VLIW	Very Large Instruction Word

BIBLIOGRAPHY

[1] Ron Wilson. (2002, September). Solutions Proposed for Verification Crisis. *EE Design, EETIMES Network.* [Online]. Available: http://www.eedesign.com/story/ OEG20020930S0054

[2] Vince Emery. (1996). The Pentium Chip Story: A Learning Experience. [Online]. Available: http://www.emery.com/1e/pentium.htm..

[3] S. I. Assn. (1999). International Technology Roadmap for Semiconductors. ITRS. [Online]. Available: http://public.itrs.net.

[4] Janick Bergeron, "Writing Testbenches, Functional Verification of HDL Models," 2nd ed. Kluwer Academic Publishers: 2003, pp. xix – xxv.

[5] Model Technology Inc. Verification Insight - Cutting Thru the Verification Confusion. Verification Seminar, 2002. [Online]. Available: http://www.mentor. com/ consulting

[6] Joan Bartlett. (2003, March). The Case for SystemC. *EEDesign, EETIMES Network.* [Online]. Available: http://www.eedesign.com/story/OEG20030307 S0020.

[7] C.-J. H. Seger, "An Introduction to Formal Verification," UBC, Department of Computer Science, Vancouver, B.C., Canada, Technical Report 92-13, June 1992.

[8] Hussain Said Al-Assad, "Lifetime Validation of Digital Systems via Fault Modeling and Test Generation," Ph.D. dissertation, University of Michigan, 1998.

[9] Y. Zan, M. Byeong, and C. Gwan, "Si-Emulation: System Verification Using Simulation and Emulation," in *Proceedings of the International Test Conference*, Atlantic City, NJ, October 2000, pp. 160-169.

[10] M. S. Gurmeet, K. Anshul, and K. Shashi, "Circuit Partitioning with Partial Order for Mixed Simulation Emulation Environment," *in the 6th IEEE International Workshop on Rapid System Prototyping (RSP),* Chapel Hill, North Carolina, June 1995, pp. 201-207.

[11] S. B. Sarmadi, G. Miremadi, G. Asadi and A. R. Ejlali, “Fast Prototyping with Co-Operation of Simulation and Emulation,” in *Proceedings of the 12th International Field Programmable Logic Conference (FPL)*, La Granada-Motte, September 2002 , pp. 12-25.

[12] T. Blank, “A Survey of Hardware Accelerators Used in Computer-Aided Design,” *IEEE Transactions on Design and Test*, vol. 1, no. 4, pp. 21--39, Aug. 1984.

[13] Quickturn, A Cadence Company. Hardware-based Verification is Necessary for Today’s Million Gate+ Designs. [Online]. Avaliable: http://www.quickturn. com/tech/tech.htm.

[14] M.C. McFarland, "Formal Verification of Sequential Hardware: A Tutorial, " *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 12, no. 5, pp. 633-654, May 1993.

[15] C. Kern and M. R. Greenstreet, “Formal Verification in Hardware Design: A Survey,” *ACM Transactions on Design Automation of Systems,* vol. 4, pp. 123-193, April 1999.

[16] Aarti Gupta, "Formal Hardware Verification Methods: A Survey, " *Formal Methods in System Design*, Vol. 1, pp. 151-238, 1992.

[17] E. Clarke and J. Wing, “Formal Methods: State of the Art and Future Directions,” CMU Computer Science, Technical Report CMU-CS-96-178, August 1996.

[18] R. P. Kurshan, "Formal Verification in a Commercial Setting," in *Proc. Design Automation Conference*, Anaheim, California, June 9-13, 1997, pp 258-262.

[19] D. Dill,"Formal Verification: Experiences and Future Prospects," presented at POPL 1999. [Online]. Available: http://www.cerc.utexas.edu/~jay/fv_surveys/.

[20] A. J. Hu, "Formal Hardware Verification with BDDs: An Introduction," in *Proc. Pacific Rim Conference on Communications, Computers and Signal Processing*, 1997, pp. 677-682.

[21] Per Bjesse, “Gate Level Description of Synchronous Hardware and Automatic Verification Based on Theorem Proving,” thesis for the Degree of Doctor of Philosophy, Dept. of Computer Science, Chalmers University of Technology and Goteborg University, May 2001, pp. 1-7.

[22] V.K. Pisini, S. Tahar, P. Curzon, O. Ait-Mohamed, and X. Song, "Formal Hardware Verification by Integrating HOL and MDG," in *GLS-VLSI'00*, Chicago, USA, 2000.

[23] K. Schneider and T. Kropf, "Verifying Hardware Correctness by Combing Theoreme Proving and Model Checking," University of Karlsruhe, Karlsruhe, Germany, Tech. Rep. SFB358-C2-5/95, December 1995.

[24] S. Rajan, N. Shankar, and M. K. Srivas, "An integration of model-checking with automated proof checking," in *Proceedings of the 7th International Workshop on Computer-Aided Verification*, Springer-Verlag 1995.

[25] J. J. Joyce, and C.H. Seger "Linking BDD-Based Symbolic Evaluation to Interactive Theorem-Proving," University of British Columbia, Computer Science Department, Technical report TR-93-18, , Vancouver, B.C, 1993.

[26] Lars Philipson and Lunds Tekniska Hogskola. (2001, November). Survey Compares Formal Verification Tools. *Exclusive Features, EEdesign, EETIMES Network*. [Online]. Available: http://www.eedesign.com/features/exclusive/ OEG20011128S0037.

[27] Farhad Mavaddat. Formal Hardware Verification: A Users' View Summary. Dept. of CS, University of Waterloo. [Online]. Available: http://asic. union. edu/Asic98/Wrkshp/verification.html.

[28] J. R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," presented at the *27th Design Automation Conference, ACM*, New York, 1990.

[29] C. Michael Chang. (2001, March). Formal Verification Moves Firmly into the Design Environment. *EEdesign, EETIMES Network*. [Online]. Available: http://www.eedesign.com/story/OEG20010301S0058.

[30] Nicolas Mokhoff. (2001, June). Intel, Motorola Report Formal Verification. *EETimes, EETIMES Network*. [Online]. Available: http://www.edtn.com/ story/OEG20010621S0080.

[31] X. Zhu. (1999, September). Presentation Slides: Event Driven Simulation, Today and Perspective. Dept. of EE, Princeton University. [Online]. Avialable: http://www.ee.princeton.edu/~xzhu/paper/eventsimu.pdf.

[32] K. Westgate and D. McInnis. Cycle-Based Simulation: Reducing Simulation Time with Cycle Simulation. Cadence. [Online]. Available: http://www.quickturn.com/tech/cbs.htm.

[33] J. Bauer, M. Bershteyn, I. Kaplan, and P. Vyedin, "A Reconfigurable Logic Machine for Fast Event-Driven Simulation," in *Proceedings of ACM/IEEE Design Automation Conf.* (DAC), 1998, pp. 668-671.

[34] J. Sanguinetti, "Language Considerations and Experimental Results Using A Verilog Compiler," presented at the *2nd Annual International Verilog HDL Conference*, 1993.

[35] Synopsys Inc.. (2001, May). Simulation Server Farms. *The Synopsys technical bulletin for design and verification engineers*, issue 1. [Online]. Available: http://www.synopsys.com/news/pubs/veritb/q101/veritb_sec3_art1.html.

[36] Synopsys Inc.. (2001, July). Synopsys and Platform Computing Cooperate to Advance Simulation Server Farm Technology. [Online]. Available: http://www.synopsys.com/news/announce/press2001/vcs_platform_pr.html.

[37] Cadence' white paper. (2002, March). The Case for A Simulation Farm. [Online]. Available: http://www.cadence.com/whitepapers/server_farm.html.

[38] N. CaÑellas and J.M. Moreno, "Speeding UP Hardware Prototyping by Incremental Simulation/Emulation," in *Proceedings of 11th International Workshop on Rapid System Prototyping (RSP)*, June 2000, Paris, France, pp. 98.

[39] S.Cadambi, C. S. Mulpuri, and P. N. Ashar, "A Fast, Inexpensive and Scalable Hardware Acceleration Techniques for Functional Simulation," in *Proceedings of ACM/IEEE Design Automation Conf.* (DAC), 2002, pp.570-575.

[40] D. Kirovski, M. Potkonjak, and L. M. Guerra, "Impoving the Observability and Controllability of Datapaths for Emulation-Based Debugging," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol 18, no.11, pp.1529, November 1999.

[41] E. M. Sentovich, D. Dill, and S. Tasiran, "Simulation Meets Formal Verification," presented at *the IEEE Intl Conf. on Computer-Aided Design, ICCAD1999,* San Jose, CA, 1999.

[42] Shipra Panda. (1999). Directed Simulation Using Trajectory Evaluation. Dept. of ECE, Carnegie Mellon University. [Online]. Available: http://citeseer.nj.nec.com/481218.html

[43] Y. C. Kim, and K. K. Saluja, "Sequential Test Generators: Past, Present and Future," *INTEGRATION, the VLSI Journal*, vol. 26, pp. 41-54, 1998.

[44] L. Nachman, K. K. Saluja, S.J. Upadhyaya, and R. Reuse, "A Novel Approach to Random Pattern Testing of Sequential Circuits," *IEEE Transactions on Computers*, vol. 47, no.1, pp. 129-134, January 1998.

[45] Priyank Kalla. (2002, August). VLSI Logic Test, Validation and Verification. Lecture Handout (ECE6960), University of Utah. [Online]. Available: http://www.ece.utah.edu/~kalla/lectures/lec1.pdf.

[46] H. Al-Assad and J. P. Hayes, "Design Verification via Simulation and Automatic Test Pattern Generation," in *International Conference on Computer-Aided Design*, 1995, pp.174-180.

[47] B.Chen, C.L. Lee, and J.E. Chen, "Design Verification by Using Universal Test Sets," in *Proc. Asian Test Symposium*,1994, pp.261-266.

[48] M.S. Abadir, J. Ferguson, and T.E.Kirkland, "Logic Design Verification via Test Generation," *IEEE TCAD*. vol.7, no.1, January 1988, pp.138-148.

[49] D. Van Campenhout, H. Al-Asaad, J. P. Hayes, T. Mudge, and R. Brown, "High-level Design Verification of Micro-processors via Error Modeling," *ACM Transactions on Design Automation of Electronic Systems* , vol.3, no.4, October 1998, pp. 581-599.

[50] David Van Campenhout, Trevor Mudge, and John P. Hayes, "High-level Test Generation for Design Verification of Pipelined Microprocessors," in *Design Automation Conference (DAC)*, 1999, pp.185-188.

[51] Robert McNaughton, "Unate Truth Functions," *IRE Trans. On Electronic Computers*, vol. EC-10, pp. 1-6, March 1961.

[52] Sheldon B. Akers, JR., "Universal Test Sets for Logic Networks," *IEEE Trans. On computers*, vol. C-22, no.9, pp.835-839, September 1973.

[53] Sudhakar M. Reddy, "Complete Test Sets for Logic Functions," *IEEE Trans. On Computers*, vol. C-22, no.11, pp.1016-1020, November 1973.

[54] F. Fallah, P. Ashar, and S. Devadas, "Simulation Vector Generation from HDL Descriptions for Observability Enhanced Statement Coverage," presented at the *36th Design Automation Conference*, New Orleans, June 1999.

[55] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, "A Functional Validation Technique: Biased-Random Simulation Guided by Observability-Based Coverage," in *International Conference on Computer Design*: VLSI in Computers &amp; Procssors (ICCD'01), September 2001, pp. 82-88.

[56] C. Norris Ip, "Simulation Coverage Enhancement Using Test Stimulus Transformation," presented at *International Conference on Computer-Aided Design (ICCAD)*, November 2000.

[57] P. Faye, E. Cerny, P. Pownall, "Improved Design Verification by Random Simulation Guided by Genetic Algorithms," *IFIP World Comp. Congress, Conf. ICDA/APChDL*, August 2000, Beijing, pp.456-466

[58] E. Bin, R. Emek, G. Shurek, and A. Ziv, "Using A Constraint Satisfaction Formulation and Solution Techniques for Random Test Program Generation," IBM *SYSTEMS JOURNALS*, vol. 41, no.3, 2002, pp.386-402.

[59] A. Gupta, A.E. Casavant, P. Ashar, X.G. Liu, A. Mukaiyama, and K. Wakabayashi, "Property-Specific Testbench Generation for Guided Simulation," in *Proceedings of the 15th International Conference on VLSI Design* (VLSID'02), January 2002, pp.524.

[60] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal, "Coverage-directed Test Generation Using Symbolic Techniques," presented at the *1st International conference on Formal Methods in Computer-aided Design*, December 1996.

[61] J. Yuan, J. Shen, J. Abraham, and A. Aziz, "On Combining Formal and Informal Verification," in Proceedings of Conference on Computer-Aided Verification, June 1997, pp.376-387.

[62] R. Allen and D. Gajski. (2000). The Case for C/C++ Hardware Design. *Technology Trends*, *EETIMES*, *EETIMES Network*. [Online]. Available: http://www.eetimes.com/special/special_issues/2000/techtrends/hardware.html.

[63] Synopsys' White Paper. (2001, April). OpenVera Technology Backgrounder. [Online]. Available: http://www.open-vera.com/technical/openvera_tb.pdf.

[64] Obsidian Software, Inc. Verification Language: Hardware Verification Language Jump Table. [Online]. Available: http://www.obsidiansoftware.com/verification-languages.htm.

[65] Peter M. Maurer. Reference Manual for a Data Generation Language Based on Probabilistic Context Free Grammars, Version 1.0. [Online]. Available: http://www.cs.odu.edu/~zeil/cs355/Handouts/dglmanual.pdf.

[66] T. Kuhn, T. Oppold, C. Schulz-Key, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai, "Object Oriented Hardware Synthesis and Verification," in *ACM, ISSS'01*, October, 2001, Montreal, Quebec, Canada. pp.189 -194.

[67] Verisity Design Inc. (2003). Open Licensing e Language. [Online]. Available: http://www.verisity.com/programs/licensee/index.html

[68] Verisity Design Inc. (2003). Driving the *e* Language towards Open Public Standardization. [Online]. Available: http://www.verisity.com/programs/licensee/partners.html

[69] Juniper Networks Inc & Jeda Technologies, Inc. Jeda Programming Language User Manual. [Online]. Available: http://www.jeda.org/manual/index.html.

[70] Libero Version 2.32, Copyright © 1996-97 iMatix. Libero Home Page. [Online]. Available: http://www.cs.vu.nl/~eliens/documents/libero/ .

[71] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H.Yang, "Protocol Verification as a Hardware Design Aid," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE Computer Society, 1992, pp. 522-525.

[72] D.L. Dill, "The Murphi Verification System," In Rajeev Alur and Thomas Henzinger, editors, Computer-Aided Verification, CAV '96, volume 1102 of Lecture Notes in Computer Science, pp. 390-- 393, New Brunswick, NJ, July/August 1996

[73] David L. Dill's research group, Stanford University. (1996, May). Murphi Description Language and Verifier. [Online]. Available: http://sprout.stanford.edu/ dill/murphi.html#Overview

[74] Forte Design Systems' White Paper. Quickbench Sequencer. [Online]. Available: http://www.forteds.com/quickbench/qbsequencer.asp

[75] Averant's White Paper. Solidify™ Data Sheet. [Online]. Available: http://www.hdac.com/Solidify_Datasheet.pdf.

[76] Averant's White Paper. (2000). Solidify-static Functional Verification for HDL Design. [Online]. Available: http://www.averant.com/solidify_ds.pdf.

[77] ON-THE-FLY, LTL MODEL CHECKING with SPIN. [Online]. Available: http://spinroot.com/spin/whatispin.html#A.

[78] P. Flake and D. Rich. (2001, February). A Practical Approach to System Verification and Hardware Design, the Public Subset of the SUPERLOG Language. Co-design Automation, Inc. [Online]. Available: http://home.btconnect.com/cdauk/tutorial/tsld001.htm

[79] R. Goering and P. Clarke. (2001, January). Superlog Design Language Picks up Speed. *EETIMES*, *EETimes Network*. [Online]. Available: http://www.eetimes.com/story/OEG20010122S0024

[80] Peter Clarke. (2001, June). Co-Design Preps Superlog for Standardization. *EETIMES*, *EETimes Network*. [Online]. Available: http://www.eetimes.com/story/OEG20010611S0102.

[81] Peter Clark. (2002, August). Synopsys Snaps up Co-Design for Superlog Language. *Semiconductor Business News*, *EETimes Network*. [Online]. Available: http://www.siliconstrategies.com/story/OEG20020828S0012.

[82] Open SystemC Initiative. Overview. [Online]. Available: http://www.systemc.org/projects/sitedocs/document/overview.

[83] J. Connell and B. Johnson. (2003). Early Hardware/Software Integration Using SystemC2.0. [Online]. Available: http://www.synopsys.com/products/cocentric_studio/esc_paper_552.pdf.

[84] Vassilios Gerousis. (2003, March). SystemVerilog 3.1, the HDVL Standard. Accellera Org. [Online]. Available: http://www.accellera.org/svintro.pdf.

[85] Faisal Haque. (2003, January). SystemVerilog 3.1 Adds Assertions and Testbench Automation. *EEdesign, EETimes Network*. [Online]. Available: http://www.eedesign.com/ features/exclusive/OEG20030110S0057

[86] W. Mueller, R. Dömer, and A. Gerstlauer, "The Formal Execution Semantics of SpecC," presented at *Proceedings of International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[87] S. Saoud, D. Gajski, and R. Dömer, "Specification and Validation of New Control Algorithms for Electric Drives Using SpecC Language," presented at *Proceedings International Conference on Systems, Man and Cybernetics*, Hammamet, Tunisia, October 2002.

[88] R. Dömer, D. Gajski, and A. Gerstlauer, "SpecC Methodology for High-Level Modeling," presented at *the 9th IEEE/DATC Electronic Design Processes Wrokshop*, Monterey, California, April 2002.

[89] The TestBuilder Team. Cadence Verification Extensions (CVE). [Online] Available: http://www.testbuilder.net/tb_systemc.thtml.

[90] Synopsys Inc. (2003, February). Constrained-Random Test Generation and Functional Coverage with Vera. [Online]. Available: http://www.synopsys.com/ products/vera/vera60_wp.pdf

[91] Richard Goering. (2001, April). Synopsys Opens Vera as Verification Language Standard. *EEdesign, EETimes Network*. [Online]. Available: http://www. eedesign.com/story/OEG20010402S0048.

[92] Chris Edwards. (2002, April). Formal Verification Push for OpenVera Language. *EE Times UK*. [Online]. Available: http://www.electronicstimes.com/ story/OEG20020410S0020.

[93] Avery Design Systems Inc. (2003). TestWizard.Testbench Automation. [Online]. Available: http://www.avery-design.com/web/Avery_TestWizard_DS012003.pdf.

[94] Forte Design Systems Inc. (2002, April). Forte Design Systems' Perspective Results Analysis Product Adds Support for OpenVera. [Online]. Available: http://www.directinsight.co.uk /news/pr87.html.

[95] Synopsys Inc. (2001, April). Synopsys' Commitment to EDA Tool Interoperability Expands to Verification. [Online]. Available: http://www. synopsys. com/partners/tapin/openvera.pdf.

[96] Synopsys Inc. (2002, February). Synopsys Delivers Commercial SystemC Simulator. [Online]. Available: http://www.synopsys.com/news/announce/ press2002/cocentric_systemC_pr.html.

[97] Verisity Design, Inc. (2002, March). Testbench Acceleration, Synthesizing e testbenches Using Verisit's e Celerator. [Online]. Available: http://www.verisity. com/resources/whitepaper/printer/acceleration.html.

[98] M. Hawana and R. Schutten. (2001, December). Testbench Design, a Systematic Approach. Synopsys Inc.. [Online]. Available: http://www.synopsys.com/sps/pdf/ paper2.pdf

[99] B. Stohr, M. Simmons, J.Geishauser. (Motorola, Munich, Germany). FlexBench: Reuse of Verification IP to increase Productivity. [Online]. Available: http://www.acm.org/sigs/sigda/Archives/ProceedingArchives/Date/papers/2002/date02/pdffiles/p3e_2.pdf

[100] Synopsys' White Paper. (2002, May). Verification Intellectual Property (IP) Modeling Architecture, Guide to Structured Development Using OpenVera, version 1.1. [Online]. Available: http://www.open-vera.com/technical/ vip_arch.pdf.

[101] Synopsys Inc. (2001, September). Synopsys Launches OpenVera Catalyst Program. [Online]. Available: http://www.open-vera.com/news/openvera_ catalyst_pr.html.

[102] F. Vabid, "The Softening of Hardware," *IEEE Computer*, vol.36, no.4, page 27-34, April 2003.

[103] S. Kumar, J. H. Aylor, B. W. Johnson, W. A.Wulf, R. D. Williams, "A Model for Exploring Hardware/Software Trade-offs and Evaluating Design Alternatives." In *Hardware/Software Co-Design and Co-Verification*, edited by Jean-Michel Berge, Oz Levia, and Jacques Rouillard. Kluwer Academic Publishers, 1997.

[104] Roman, G., et al., "A Total System Design Framework," *IEEE Computer*, pp. 15-26, May 1984.

[105] Smith, C. U., L. G. Williams, "Software Performance Engineering: A Case Study including Performance Comparison with Design Alternative," *IEEE Transactions on Software Engineering*, vol.19, pp.720-741, July 1993.

[106] C. Passerone, L. Lavagno, C. Sansoe, M. Chiodo, A. Sangiovanni-Vincentelli, "Trade off Evaluation in Embedded System Design via Co-simulation," in *Proc. of ASP-DAC*, pp. 291-297, 1997.

[107] J. Blyler, "System-level Design Continues to Evolve," *Wireless Systems Design*, pp. 23-25, July/August 2002.

[108] J. Henkei, X. S. Hu, S. S. Bhattacharyya, "Guest Editors' Introduction: Taking on the Embedded System Design Challenge," *IEEE Computer*, vol. 36, no. 4, pp.35-37, April 2003.

[109] Rabi Mahapatra. (2001, September). Hardware-Software Codesign: Issues & Challenges. Texas A&M University. [Online]. Available: http://research.cs.tamu.edu/codesign/papers/681seminar2001.ppt

[110] M. Serra and W. B. Gardner, "Hardware/Software Codesign –introducing an interdisciplinary course," presented at *WCCCE Conference*, Vancouver, 1998.

[111] A. Shah and A.K. Ramani. (2000). A Survey of Co-Verification Approaches in Mixed Hardware-Software Co-Design. IIT Delhi. [Online]. Available: http://www.cse.iitd.ernet.in/~ashah/vdtt.doc.

[112] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal on Computer Simulation*, vol. 4, pp. 155--182, 1991.

[113] R. Klein and S. Leef. (1996, June). New technology Links Hardware and Software Simulators. *Electronic Engineering Times*. [Online]. Available: http://www.mentor.com/seamless/articles/eet060396.html

[114] H. Wojtkowiak. Hardware/Software Codesign. *Universität Siegen, Technische Informatik*. [Online]. Available: http://www.ti.et-inf.unisiegen.de/Forschung/Codesign/infoe.html.

[115] James A. Rowson, "Hardware/Software Co-Simulation," in *Proc. of Design Automation Conf.*, June 1994, pp 439-440.

[116] Rabi Mahapatra. (Spring 2003). Introduction to Co-simulation: Hardware-Software Co-design of Embedded Systems. Texas A&M University. [Online]. Available: http://faculty.cs.tamu.edu/rabi/cpsc689/lectures/introduction_cosimulation.pdf

[117] A. Amory, F. Moraes, L. Oliveira, N. Calzazns, and F. Hessel, "A Heterogeneous and Distributed Co-Simulation Environment, " in *15th Symposium on Integrated Circuits and Systems Design*, September 2002, pp. 115.

[118] Hessel, F., "Concepção de Sistemas Heterogêneos Multi-Linguagens", Jornada de Atualização em Informática–JAI, XXI Congresso da Sociedade Brasileira de Computação, 2001.

[119] Heiko Hübert. (June 1998). A Survey of HW/SW Co-simulation Techniques and Tools. Royal Institute of Technology, Stockholm, Sweden, TRITA-ESD-1998-07. [Online]. Available: http://citeseer.nj.nec.com/hubert98survey.html

[120] P. Dreike and J. McCoy. (June 1997). Co-Simulating Software and Hardware in Embedded Systems. *Embedded Systems Programming* [Online]. 10(6). Available: http://www.embedded.com/97/feat9706.htm.

[121] L. Semeria and A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++," presented at *IEEE International High Level Design Validation and Test Workshop (HLDVT'99)*. San Diego, CA. December, 1999.

[122] B. Tabbarra, E. Eilippi, L. Lavagno, and A. Sangiovanni-Vincentelli, "Fast Hardware-Software Co-simulation Using VHDL Models," DAC 98. [Online]. Available: http://www-cad.eecs.berkeley.edu/~polis/paper/1999/date99.pdf

[123] Michael Keating and Pierre Bricaud, "Reuse Methodology Manual for System-on-Chip Designs," Kluwer Academic Publishers, 1998.

[124] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, "Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator," in *Sixth International Workshop on Hardware/Software Codesign*, Seattle, Washington, March 1998, pp. 65-69.

[125] M. Bauer and W. Ecker, "Hardware/Software Co-simulation in a VHDL Based Test Bench Approach," in *Proceedings of the 34th DAC,* June 1997, pp.774.

[126] R. K. Gupta, C. Coelho, and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," in *Proceedings of 29th ACM/IEEE Design Automation Conference*, June 1992, pp. 129-134.

[127] D. Becker, R. K. Singh, and S. G. Tell, "An Engineering Environment for Hardware/Software Co-simulation," in *Proceedings of 29th ACM/IEEE Design Automation Conference*, June 1992, pp.129-134.

[128] J.P.Soinenen, T.Huttunen, K.Tiensyrija, and H.Heusala, "Co-simulation of Real Time Control Systems," in *European Design Automation Conference (EURODAC)*, September,1995, pp.170.

[129] Y. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration," in *Proc. ACM/IEEE Design Automation Conf.*, June 1995, pp. 456-461.

[130] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient Software Performance Estimation Methods for Hardware/Software Codesign," in *Proc. ACM/IEEE Design Automation Conf.*, June 1996, pp.605-610.

[131] K. Hines and G. Borriello, "A Geographically Distributed Framework for Embedded System Design and Validation," in *Proc. ACM/IEEE Design Automation Conf.*, June 1998, pp.140-145.

[132] M. Dalpasso, A. Bogliolo, and L. Benini, "Virtual Simulation of Distributed IP-based Designs," in *Proc. ACM/IEEE Design Automation Conf.*, June 1999, pp. 50-55.

[133] R. Goering. (1999, Jan). Global Chip Design Raises Promises and Challenges. *EE Times*. [Online]. Available: http://www.eetimes.com/story/OEG19990111S0016.

[134] H. Lavana. A. Khetawat, F. Brglex, and K. Kozminski, "Executable workflows: A Paradigm for Collaborative Design on the Internet," in *Proc. ACM/IEEE Design Automation Conf.* June 1997, pp 533-558.

[135] Braudio Adriano De mello and Flavio Rech Wagner. (2001, December). A Standardized Co-simulation Backbone. *VLSI-SOC 2001*. pp.181-192. [Online]. Available: http://www.urisan.tche.br/~bmello/trab-pub/forum-pirenopolis.pdf

[136] F. Kuhl, J. Dahmann, R. Weatherly, "Creating Computer Simulation Systems: An Introduction to the High Level Architecture," Prentice Hall, Paperback, BK&Cd Rom edition, October 1999.

[137] Pitch Kunskapsutveckling AB, Linköping, Sweden. High Level Architecture for Beginners. PITCH Knowledge Network. [Online]. Available: http://www.pitch.se/hla/hlaforbeginners.asp

[138] S. Straßburger, T. Schulze, U. Klein, and J. O. Henriksen. (December, 1998). Internet-based Simulation Using Off-the-shelf Simulation Tools and HLA. Proceedings of WSC, Washington, USA. [Online]. Available: http://citeseer.nj.nec.com/355794.html.

[139] Sungjoo Yoo, Kiyoung Choi, and Dong Sam Ha, "Performance Improvement of Geographically Distributed Cosimulation by Hierarchically Grouped Messages," *IEEE Transactions of VLSI Systems*, vol. 8, no.5, pp. 492-502,October, 2000.

[140] S. Yoo and K. Choi, "Optimizing Timed Cosimulation by Hybrid Synchronization," Design Automation for Embedded Systems. Norwell, MA: Kluwer Academic, June 2000, vol. 5, pp. 129-152.

[141] M. Chetlur and N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wislsey, "Optimizing Communication in Time-Warp Simulators," in *Proc. 12th Workshop Parallel and Distributed Simulation*, May 1998, pp.64-71.

[142] K. Hines and G. Borriello, "Optimizing Communication in Embedded System Co-simulation," in *Proc. Int. Workshop Hardware-Software Codesign*, March 1997, pp.121-125.

[143] S. Yoo and K. Choi, "Synchronization Overhead Reduction in Timed Cosimulation," in *Proc. IEEE Int. High Level Design Validation and Test Workshop*, Nov.1997, pp.157-164.

[144] UC Berkeley, EECS, CA. The Ptolemy Project: Heterogeneous Modeling and Design. [Online]. Available: http://tolemy.eecs.berkeley.edu/

[145] A. Kalavade and E. A. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp.16-28, September 1993.

[146] UC Berkeley, EECS, CA. The Almagest – Volume 1, Ptolemy 0.7 User's Manual. [Online]. Available: http://ptolemy.eecs.berkeley.edu/ptolemyclassic/almagest/user.htm

[147] B. L. Evans, A. Kamas, and E. A. Lee, "Design and Simulation of Heterogeneous Systems Using Ptolemy," in *Proceedings of the 1st Annual Conference of the Rapid Prototyping of Application Specific Signal Processors (RASSP) Program*,1994, pp. 97-105.

[148] UC Berkeley, EECS, CA. A Framework for Hardware-Software Co-Design of Embedded Systems. [Online]. Available: http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html

[149] Sushant Jain and Vivek Sinha. (1998, December). System Synthesis Using Public Domain Tools: Survey of Embedded System Design Tools. Dept. of Computer Science & Engineering, Indian Institute of Technology, Delhi. [Online]. Available:http://www.cse.iitd.ernet.in/esproject/docs/projects/sushant_vivek/sem1/end_term/report/rep/rep.html

[150] Mentor Graphics Corp., Wilsonville. (1996). Seamless Co-Verification Environment, User's Reference Manual.

[148] Russel Klein, Serge Leef. (1996, June). New Technology Links Hardware and Software Simulators. Electronic Engineering Times. [Online]. Available: http://www.mentor.com/seamless/articles/eet060396.html

[152] Giampaolo Figini. Seamless Co=Verification Environment™ from Mentor Graphics. [Online]. Available: http://eda.sci.univr.it/~agliada/menu_frame_left_file/01/page_file/Documents/Articles/seamless.htm

[153] Mentor Graphics Corp. Seamless Hardware/Software Co-Verification: Software Debugger, Hardware Simulator, and Platform Support for v4.2 Seamless CVE PSPs. [Online]. Available: http://www.mentor.com/seamless/psp_listings.html.

[154] Gabe Moretti.(2002, April). Technology Provides a Bridge to C. EDN. E-insite Network. (Online). Available: http://www.einsite.net/ednmag/index.asp?layout=articlePrint&articleID=CA209110.

[155] Richard Goering. (2000, November). Co-design Tools Draw a Diverse Crowd. *EE Times*. [Online]. Available:http://www.eetimes.com/story/OEG20001129S0018.

[156] John Cooley. (2001, July). Subject: Mentor 'Platform Express', Seamless, Synopsys Eagle-I, Summit VCPU. ESNUG, DEEPCHIP. [Online]. Available: http:www.deepchip.com/items/dac01-10.html.