A Multi-Abstraction Level Platform for the Validation and Verification of Complex Digital Designs

Jean-François Boland

Doctor of Philosophy

Electrical and Computer Engineering

McGill University

Montreal,Quebec

2007-02-07

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy

DEDICATION

To my parents and Marie-Hélène

ACKNOWLEDGMENTS

After all the years and effort, it seems that my student "career" is finally coming to an end. It is a wonderful feeling when I think of all the people who made this work possible.

First and foremost, I wish to thank my advisers, Professor Zeljko Zilic and Professor Claude Thibeault for their support and encouragement throughout my doctoral studies. While allowing me considerable freedom to conduct research on my own, Dr. Zilic taught me the skills to successfully formulate and approach a research problem. Dr. Thibeault gave me the chance to work with a highly qualified team on the PROMPT project. His advice and precious suggestions helped steer this work in the right direction.

I would also want to thank Professor François Gagnon for his encouragement. He was the first person to incite me to start my Ph.D. studies. I would like to thank Professor Nicholas Rumin for taking precious time to serve on my thesis committee. His wisdom and great mind have had a positive influence on me.

I am also grateful to the MAME project team for their helpful contributions. Without their experience and help, I would not have been able to conduct experiments on real complex designs. I keep very good memories of the many constructive brainstorming sessions with Dr. Yvon Savaria and Alexandre Chureau. I am also thankful to my office-mates, Henry Chan, Stephan Bourduas, and Françis Beaudoin for their pleasant company. I had great time with them and they make the life as a graduate student much more enjoyable and unforgettable.

I thank the Regroupement Stratégique de Microéletronique du Québec (ResMiQ) for its generosity in providing financial support for my travels to present my research results at international conferences. Special thanks go to the university École de Technologie Supérieure for their trust in me. They provided me with funding support throughout my stay at McGill, so I could spend my entire time on my research activities.

Special thanks go to all the undergrad students which I taught during my master's degree. You were the main reason for me to do a PhD, so I could come back later as a professor and continue a stimulating career. Without you all, I would not have been so perseverant during all those years. I also appreciate many of my personal friends who always keep faith in me. Thank you all for those tremendous party nights who help me keep a good balance between my professional hard work and the pleasure of life.

I thank my parents for always being supportive and encouraging in my pursuit of academic excellence. I would like to express my deepest gratitude for their constant care and love. And the last but not the least, I would like to thank my girlfriend Marie-Hélène who has shaped my life more than anyone else. We met at the beginning of my Ph.D. and she stays always close to me, providing constant support, comprehension, and motivation. She is an extraordinary person with always the might words for both good times and bad. Without her, I do not think I would ever have completed this dissertation.

ABSTRACT

Design verification is one of the most challenging tasks in hardware development. With the ever increasing complexity of digital systems, validation and verification have become the primary bottleneck in circuit design, consuming up to 70% of the total effort in a project. Most of this time is spent on testbench creation and debugging. Complex digital system design is a process that spreads across multiple abstraction levels, using various software tools and languages.

This dissertation addresses the validation and verification problem using a unified approach, which utilizes new mechanisms to bridge the gap between abstraction levels and provides a new simulation-based verification methodology. The goal is to reduce the time spent on verification while increasing the testbench quality.

The first mechanism is a cosimulation interface between MATLAB/Simulink and SystemC called SimSyC. The goal of SimSyC is to bridge the abstraction gap that exists between the algorithmic level and the lower levels. By providing a configurable communication link between MATLAB/Simulink and SystemC, it is now possible to reuse high abstraction level models to validate and verify system-level and register transfer-level representations. The time spent on testbench development is considerably reduced by reusing Simulink's data generators and data analysis modules. The pre-verified building blocksets library in Simulink dramatically increases the quality and the efficiency of the testbench. Moreover, SimSyC opens up a wide range of visualization and data analysis capabilities to the SystemC simulation kernel. Experiments on three case studies have shown that SimSyC provides between

one and two orders of magnitude speedup for testbench development and enables verification strategies that were simply not possible before.

Secondly, we present a generalized version of the transactor used for transaction-based verification. Called segmented adapter, this novel mechanism addresses the migration problem inherent to transactors. Our proposed partitioning into three specific segments provides the modularity necessary for reuse and migration across abstraction levels and projects. A SystemC realization is proposed for these segmented transactors. We demonstrate the capabilities and efficiency of segmented adapters through the validation and verification of a multi-equalizer design successively refined across three abstraction levels.

These two mechanisms are the foundation of a multi-abstraction level verification methodology integrated within a framework of abstraction-refinement based design. This methodology promotes early verification and vertical testbench reuse across abstraction levels. The verification platform that we have developed in this thesis has proven to be a valuable addition to the range of simulation-based methods already available.

ABRÉGÉ

La validation et la vérification de circuits numériques sont aujourd'hui considérées comme les plus imposants défis que doivent relever les ingénieurs de conception. D'une complexité toujours croissante, les circuits numériques sont devenus depuis quelques années des systèmes entiers sur puce de silicium. Composés de plusieurs fonctions et d'une capacité de calcul phénoménale, la vérification de ces systèmes-sur-puce peut consommer près de 70% du temps total alloué à un projet. La conception de systèmes numériques complexes exige une modélisation à plusieurs niveaux d'abstraction, ce qui exige l'utilisation de plusieurs langages et outils de conception. Ceci a pour conséquence d'alourdir considérablement le développement d'environnement de vérification fiable et efficace.

Cette dissertation concerne la problématique de la validation et la vérification de systèmes numériques complexes. De nouveaux mécanismes pour concilier les niveaux d'abstraction ainsi qu'une méthodologie de vérification basée sur la simulation sont développés dans le but de réduire le temps associé à la vérification tout en augmentant la qualité de cette dernière.

Le premier mécanisme développé est une interface de cosimulation entre MATLAB/Simulink et SystemC appelée SimSyC. Le but premier de cette interface est de relier le niveau d'abstraction algorithmique aux niveaux inférieurs. De cette façon, il est possible de réutiliser les modèles haut niveau pour la validation et la vérification des modèles au niveau de l'architecture du système ou du transfert des registres. Le temps de développement des bancs d'essais est considérablement réduit par la

réutilisation des modules d'analyse et de génération de données. L'utilisation de la librairie de blocs pré-vérifiés de Simulink augmente la qualité et l'efficacité du banc d'essais. De plus, SimSyC permet d'utiliser les librairies spécialisées ainsi que l'interface graphique de Simulink pour analyser de façon plus efficace le comportement d'un modèle SystemC. Les résultats expérimentaux de trois études de cas démontrent que le temps consacré à l'élaboration des bancs de test est réduit d'un facteur pouvant varier de un à deux ordres de grandeur. De plus, SimSyC permet d'élaborer une stratégie de vérification à multi-niveaux d'abstraction jusqu'ici irréalisable.

Dans un deuxième temps, une version généralisée des transateurs est présentée. Appelé adaptateur segmenté, ce mécanisme nouveau genre offre une solution innovatrice au problème de migration des transacteur traditionnels. Nous proposons un partitionnement du transacteur en trois segments distincts pour ainsi permettre une réutilisation partielle de ce dernier pour pouvoir le migrer facilement d'un niveau d'abstraction à un autre. La réalisation d'adaptateurs segmentés en langage SystemC est aussi démontrée dans le cadre d'une étude de cas portant sur la validation et la vérification d'un multi-équaliseur développé par raffinements successif sur trois niveaux d'abstractions.

Ces deux mécanismes forment la base d'une méthodologie de validation et de vérification sur plusieurs niveaux d'abstraction, intégrée à un processus de conception par raffinement successif. Les éléments clef de cette méthodologie sont la vérification hâtive du modèle du système ainsi que la réutilisation verticale du banc de test d'un niveau d'abstraction à l'autre. La plateforme de vérification développée par

ces travaux de recherche a prouvée être une addition substantielle aux méthodes et outils de vérification déjà disponibles.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

Over the last 40 years, integrated circuit (IC) complexity has increased drastically. By complexity, we refer here to the number of transistors that can be integrated on a single chip, regardless of what they do. The well known Moore's law [42] predicts that this number is doubling every 18 months. Started by Jack Kilby in 1958 with a single transistor, today's integrated circuits contain up to 1 million transistors per mm^2. These thoughts are stimulated by the latest update of the International Technology Roadmap for Semiconductors published by the Semiconductors Industry Association (SIA) [61]. This report indicates that integrated circuits will continue to become smaller and denser through the year 2020.

With this increased number of transistors, design size grows as more functionalities are being integrated onto a single chip. Microprocessors, memory, digital signal processors, programmable logic and custom logic, just to name a few, are now integrated onto a single chip to form systems-on-a-chip (SoC). Nevertheless, tools and methodologies used to design these complex systems have not advanced as fast as fabrication. A design gap began to appear quickly between the number of transistors that can be integrated on a single chip and the number that can be used in a design. The Electronic Design Automation (EDA) industry has contributed significantly over the last two decades to shortening this gap and maximizing hardware resources usage. Having more transistors available than what we are able to handle

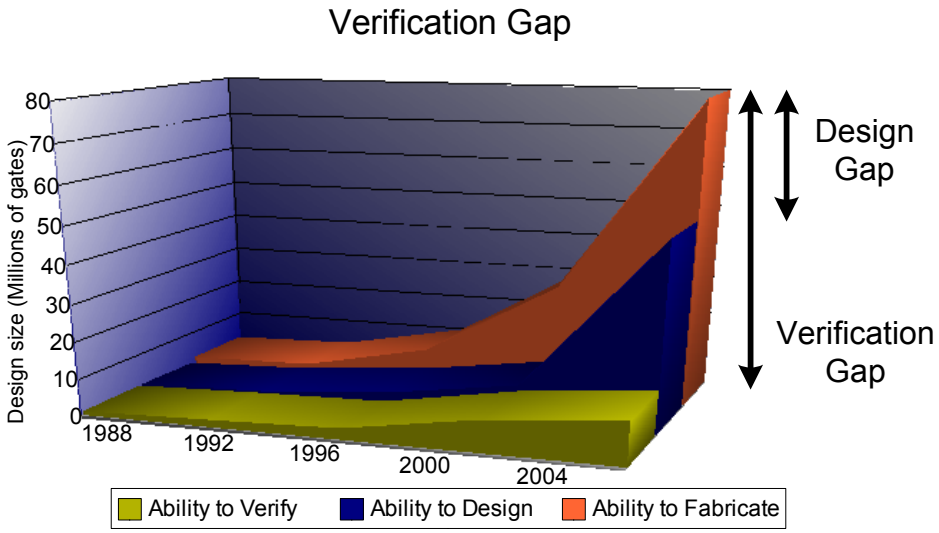

Figure 1–1: Verification gap [6]

is a problem with relatively moderated consequences. These transistors left unused do not interfere with the design performance and quality. A more critical problem faced by IC designers concerns the validation and verification of these complex digital systems. Design validation and verification (V&V) is the process of ensuring correctness of the design throughout the design stages. Verification complexity is growing proportionately with the square of the increase in design complexity [6]. With next-generation process technologies of 0.06 μm capable of supporting beyond 100 million transistors per mm^2, the verification problem is only getting worse over the years. In a similar way to what we observed with the ability to fabricate, there is a more important gap between the ability to design and the ability to verify. However, the consequences here are much more risky. Instead of unused transistors, we are talking about partially verified designs. IC performance and quality are now

directly affected. Designer productivity and design possibilities are severely reduced by this verification gap. Figure 1–1 summarizes graphically the integrated circuit design reality.

This dissertation is concerned with the verification of complex digital systems. These systems are described with a variety of tools and languages at different levels of abstraction, resulting in complex verification problems. Software Defined Radio (SDR) is a good example of such systems. It includes hardware and software components that require rigorous verification all along the design flow. The verification requirements of such systems become quickly more challenging than the design itself. SDR design usually begins at a high-level of abstraction and it is being refined down to lower levels. This causes problems and discontinuities with the verification environment that have to be adapted at each abstraction level.

It is widely accepted that functional verification is the most imposing obstacle to meeting time-to-market schedules [4, 32]. Common industry estimates are that functional verification constitutes near 70% of the total effort on a project. With IC getting bigger and more complex, the verification process is more time-consuming and expensive [53]. Design size has a dramatic effect on controllability and observability of the design. Because they scale inversely with design complexity, more tests need to be created to reach some internal states of the design. For example, if a design complexity doubles, observability will half and controllability will also be reduced by half, resulting in verification efforts that are around four times as difficult. New scalable verification solutions that handle multiple levels of design abstraction are required to cut verification time.

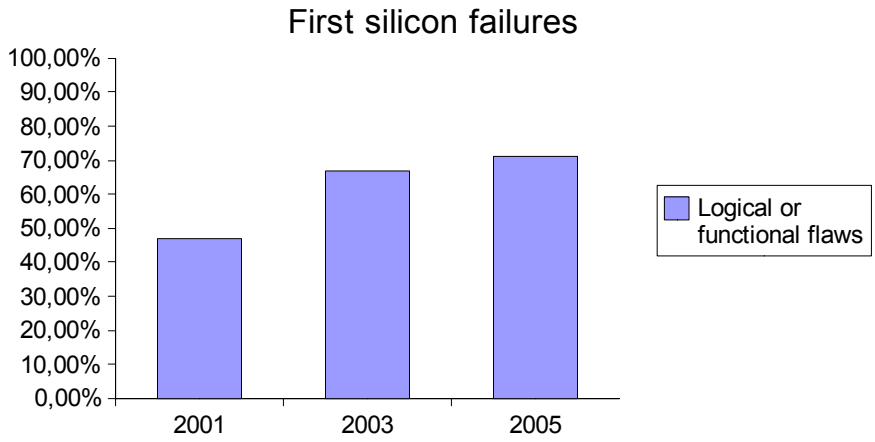

Figure 1–2: First silicon IC failure (source: Collett International Research)

Despite all the time and money that are concentrated on design verification, first-silicon fabrication success is still difficult to achieve. According to a recent survey by Collett International Research [52], 71% of all IC designs contains logic or functional flaws on first silicon fabrication. These designs thus require at least one costly silicon re-spin. Around 60% of these faulty designs have functional errors that could certainly have been detected with a more appropriate verification solution. Incorrect or incomplete specifications, corner cases simply not covered during verification or changes in design specifications are a few causes of these flaws. Figure 1–2 contrasts 2005 data to previous surveys done in 2001 and 2003. We clearly see that the percentage of faulty circuits has increased from 47% in 2001 to 71% in 2005. Detecting flaws this late in the design cycle is expensive. According to Maxfield and

Edson [40], it is an order of magnitude more expensive to fix a design problem for each delay introduced in its detection and correction.

Integrated circuit design complexity has led to a verification crisis. Engineers can no longer manually write testbenches that cover all of the possible corner-case behaviors. There are simply too many cases and it is hardly feasible to even imagine all of them. The research presented in this dissertation focuses on the development of a new technique to supplement traditional functional verification methods. The goal is to provide new mechanisms to stimulate the design with real world scenarios to verify precisely and quickly that the system operates as expected. It is important to perform this verification as early as possible in the design flow to reduce the cost of finding errors.

Traditional methods of verification have proven to be insufficient for complex digital systems. Register transfer level testbenches have become too complex to manage and slow to execute. New methods and verification techniques began to emerge over the past few years. High-level testbenches, assertion-based verification, formal methods, hardware verification languages are just a few examples of the intense research activities driving the verification domain.

The goal of this research is to develop new mechanisms to create a multi-abstraction level platform that will be used for the validation and verification of complex digital systems.

1.1 Main contributions

The main contributions of this dissertation are:

- A cosimulation interface between MATLAB/Simulink and SystemC is presented. This interface is principally used for the verification of lower abstraction level designs with a high level model of the design environment. Our contribution can also be used for a wide range of applications. For example (non exhaustive list):
  - Data visualization and analysis for SystemC models
  - Distributed model simulation
  - Heterogeneous prototyping
- An evolved version of transactors is presented. This dissertation tackles the problem of transactor migration in transaction-based verification systems. Transactor migration is defined as the capability to reuse transactors across abstraction levels with minimal changes.
- A verification methodology based on SystemC is proposed. The MATLAB/Simulink to SystemC interface and the evolved version of transactors are combined in a scalable multi-abstraction level verification platform.

All the work in this research has been published in four conference papers with a reviewing committee. The paper presented at DVCon 2005 received the best paper award.

1. A. Chureau, J.F. Boland, C. Thibeault, Y. Savaria, F. Gagnon, Z. Zilic, "Building Heterogeneous Functional Prototypes Using Articulated Interfaces", Proc. of 4th Northeast Workshop on Circuits and Systems, Gatineau, Quebec, Canada, June 2006.

2. J.F. Boland, C. Thibeault, Z. Zilic, "Using MATLAB and Simulink in a SystemC Verification Environment", Proc. of Design and Verification Conference & Exhibition, San Jose, Californie, February 2005, Best Paper Award.
3. J.F. Boland, C. Thibeault, Z. Zilic, "Efficient Multi-Abstraction Level Functional Verification Methodology for DSP Applications", Proc. of Global Signal Processing Expo, Santa Clara, Californie, September 2004.
4. J.F. Boland, A. Chureau, C. Thibeault, Y. Savaria, F. Gagnon, Z. Zilic, "An Efficient Methodology for Design and Verification of an Equalizer for a Software Defined Radio", Proc. of 2nd Northeast Workshop on Circuits and Systems, Montreal, Quebec, Canada, June 2004, pp. 73-76.

I was also invited to present the SimSyC interface at the 2nd North American SystemC Users Group meeting (2nd NASCUG), held in September 2004 at the Santa Clara convention center in California.

1.2 Organization of the thesis

This thesis is organized into six chapters. Chapter two presents background information on validation and verification of complex digital systems. Also presented is related work and current approaches to V&V. Chapter three presents a novel cosimulation interface used for multi-abstraction level verification. Chapter four presents an improved transaction-level verification method. Chapter five presents two case studies and demonstrates the performances and usefulness of the verification platform presented in this work. Chapter six concludes with results of the research and future directions.

Chapter 2
Background and related work

This chapter provides background information to clearly assess the worthiness of this research. In the following sections, the key concepts of validation and verification will be presented. The terminology used in this work will be defined and a succinct review of the related work will be presented. This review presents briefly other commercially available verification tools to put our contributions into the verification context.

2.1 Validation and Verification of complex digital systems

The words validation and verification are frequently used alternatively to express the same thing. However, there is a subtle difference between the two terms to mean two different types of analysis. According to the IEEE Standard Glossary of Software Engineering Terminology, verification is defined as "The process of evaluating a system or component to determine whether the product of a given development phase satisfy the conditions imposed at the start of that phase." On the other hand, validation is defined as "The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." In other words, V&V attempt to answer to two following fundamental questions:

- Validation: Did we build the right product?
- Verification: Did we build the product right?

In this work, we consider validation and verification as two distinct activities applied to find and correct errors in designs. Each of these activities has specific requirements and strategies to find problems in designs [35]. For example RTL design verification requires a testbench capable of creating input stimuli to verify if the design architecture behaves correctly. The same design can also be validated according to the system specifications to confirm that the design is doing the right thing. Validation usually requires a complete model of the environment to create real world validation scenarios.

Another important distinction must be done between testing and verification of digital systems [62]. In the literature, both words are often used alternatively to refer to verification activities. However, careful research indicates two distinctive disciplines. Testing usually refers to the activities involved in the process of finding errors and faults in the integrated circuit after the fabrication step. A physical chip is connected to an automatic test equipment (ATE) and a program sends test vectors to the device. Verification on the other hand uses models of the design and applies simulation-based or formal methods to verify the correctness of the implementation. In this circumstance, the word test refers to the testbench program used to verify the design. This thesis is about simulation-based verification.

2.1.1 Verification Challenges

With today's multi-million gate devices coupled with the desire to achieve faster time-to-market, verification engineers have to perform exponentially more complex verification tasks in a shorter time [33]. The success of a functional verification project relies on tools, languages, and methodologies that try to overcome a number

of challenges [5]. These challenges broadly consist of the following [29]:

Verification productivity - is the capability with which verification engineers can deal with increasingly larger designs with regard to time-to-market window. This means developing complex testbenches to stimulate the design under verification in short time period. One way to address this challenge is to move the verification environment to a higher level of abstraction.

Verification efficiency - is the skillfulness in avoiding wasted time and effort to complete the verification task. With the increasing complexity of designs, human intervention is frequently the most important factor that causes a waste of time. Therefore, new mechanisms and techniques are required to automate parts of the verification process.

Verification reusability - can be seen in two ways. (1) Capacity of components in a verification environment to be reused in new projects. (2) Ability to reuse verification components, like testbenches, across abstraction levels in the same project. Both definitions imply a modular and flexible verification environment.

Verification completeness - is a measure of the proportion of the design that has been verified. This issue is one of the most difficult one with today's complex designs. How to know when to stop? Some tools offer ways of measuring verification

progress through code coverage and other techniques [16].

This dissertation explores new methods and mechanisms to address the first three challenges. Nevertheless, improving verification productivity, efficiency and reusability will give more time to the verification engineer to focus on verification completeness.

2.1.2 Verification Methods

The verification crisis faced by the industry and the lack of complete and efficient verification solutions have stimulated research activities in this domain. To tackle this complex problem, several verification techniques have been developed over time. The result is a variety of approaches, based on different languages and tools. None of them can pretend to solve the verification problem. They have to be combined together in a verification plan to obtain good results. Figure 2–1 summarizes the principal verification techniques according to two categories: simulation-based and formal method-based [32, 4]. Literature also uses dynamic and static verification when referring respectively to simulation-based and formal method-based verification [48]. Methodologies using formal methods are said to be output driven. There is no need for input stimuli. The verification engineer specifies the output behaviors expected from the design and lets the formal checker prove or disprove it. On the other hand, methodologies using simulation runs rely on input stimuli and are then input driven. The simulation-based functional verification process typically proceeds by creating generators to produce the simulation stimuli and a simulation monitor to verify the correctness of the circuit responses [13]. To date, simulation remains

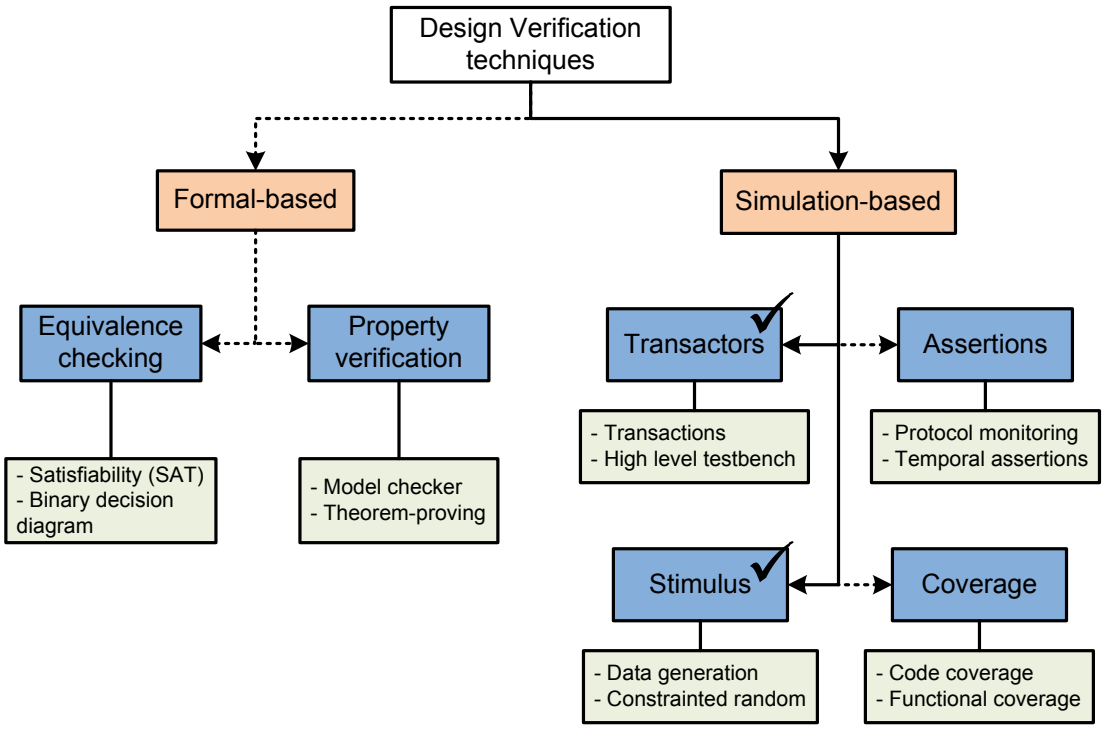

Figure 2–1: Verification techniques

the most popular method of verification to find functional design errors [60, 4]. This research concentrates on improving simulation-based verification. More precisely, we develop new mechanisms for transaction-based verification and stimuli generation. The check marks in Figure 2–1 indicate our research focus and the dashed lines represents the other verification techniques available.

2.1.3 Evolution of Simulation-based Verification Environments

To clearly understand the purpose of this research, it is important to present how the verification of digital systems has quickly evolved over the past few years. Like the evolution of design entry from schematic based tools to hardware description languages (HDL), verification has also changed in a similar way [30]. Figure 2–2

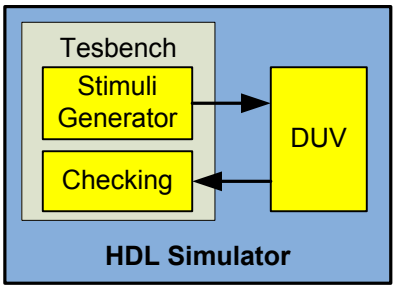

Figure 2–2: Verification environment with HDL

shows a typical verification system configuration used in the mid eighties for relatively simple designs. The design under verification (DUV) is modeled using a low level HDL language like Verilog or VHDL. The testbench consists of directed test cases, written with the same language as the design. Both programs run inside the HDL simulator and the DUV is stimulated with the test cases. This approach is sufficient for small designs which have a limited number of test cases. As design complexity increases, the number of test cases simply explodes and becomes unmanageable. Studies show that for every line of HDL used in a design, verification engineers have to write approximately five lines of HDL testbench code (or even more). In

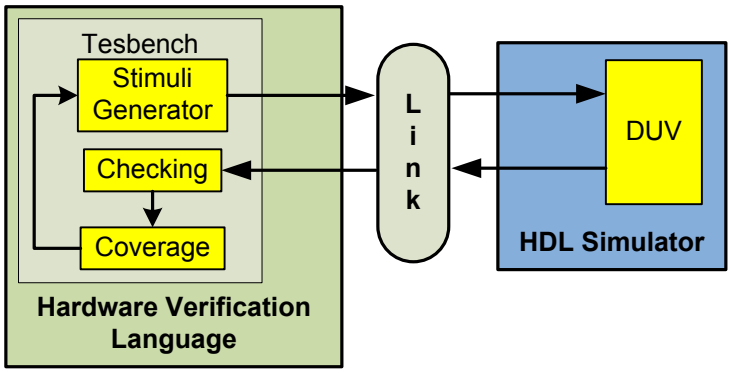

Figure 2–3: Verification environment with HVL

the mid nineties, proprietary Hardware Verification Languages (HVLs), like Vera [27], came to the rescue. These languages provided powerful constructs to simplify and accelerate the verification process. The verification code is then written using HVLs and runs in an independent simulator attached to the HDL simulator, as shown in figure 2–3. HVLs also add constrained-random test generation to the verification system. This technique enables automatic test case generation; reducing the amount of work to be done manually. A functional coverage module completes this verification system, by providing a measure for the quality of the verification. With this information, it is then possible to adjust the stimuli generator to orient the testbench toward uncovered scenarios.

The impact on verification productivity is considerable. Figure 2–4 compares the time required to achieve verification goals using HDL and HVL. With HDL, test cases are written manually, one at a time. The verification phase progresses slowly toward the goal in a staircase manner. On the other hand, HVL requires a certain amount of time to develop the constrained-random testbench. This extra time is well invested, since testcases will then be generated automatically and verification will progress more rapidly. The result is a welcome productivity gain.

Hardware verification languages are part of most verification environments for complex digital systems. However, according to the Collett International Research survey presented in chapter one [52], there is a lot of faulty integrated circuits fabricated, considering that almost 70% of the total effort on a design project is spent on verification. Digital design involves many software tools and modeling languages. The design process does not start with RTL languages anymore. High abstraction

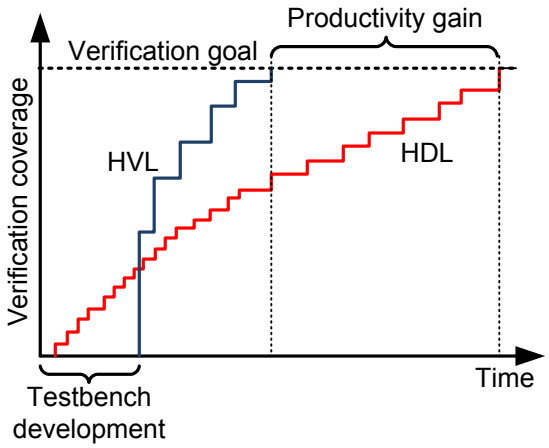

Figure 2–4: Verification productivity gain with HVLs [30]

models are created first and refined down to the RTL level. Electronic System Level (ESL) is the new level of abstraction used for complex digital systems [43]. The verification environment has to move up the same way. As discussed in chapter one, verification must begin as early as possible in the design process to find errors before they become too costly to change. One way to achieve this goal is to move some parts of the functional verification environment upward to a higher level of abstraction, while maintaining its connection to lower levels.

2.2 Multi-Level Simulation

Multi-level co-simulation has a great potential to efficiently simulate large systems containing hardware and software components, with portions of the system described at different levels of abstraction. With the growing availability of powerful parallel processing machines, parallel co-simulation is a viable approach to speed up the simulation of large systems. This section explores the important elements of a mixed-level simulation environment.

2.2.1 Programming Paradigms

In computer science, a programming paradigm refers to the view that a programmer has of the execution of the program [67]. Different programming languages implies multiples programming paradigms. Most of the programming languages support multiple paradigms. For example, the C++ programming language can be used for procedural programming, object-oriented programming, object-based programming and generic programming [67]. Other languages are designed to support only one particular paradigm. Java, for example, is designed to support object-oriented programming.

T3he design of complex digital designs is performed at multiple levels of abstraction using different programming languages. Therefore, multiple programming paradigms are used to create models of the system. When it comes to the verification of these models, problems arise when trying to make them to communicate together. We will see in chapter 3 that one way to help reducing the verification effort is to create interconnection mechanisms between abstraction levels. Representation of time and data type format are examples of important differences that exist between programming paradigms. These differences are posing challenges for the unification of abstraction levels for verification. In this work, we will have to manage the paradigm differences between MATLAB/Simulink and SystemC. The former belonging to the data flow and visual programming paradigms while SystemC support mainly the object oriented programming paradigms.

2.2.2 Abstraction Levels

To handle the increasing complexity, the design process of digital systems uses multiple levels of abstraction to represent the system. It is important at this point to define these levels. In the literature, abstraction levels are presented differently according to the application domain. They are also often separated in multiple sub-abstraction levels. Figure 2–5 gives a typical abstraction level division commonly used for hardware designs. Since there is no standard concerning the terminology

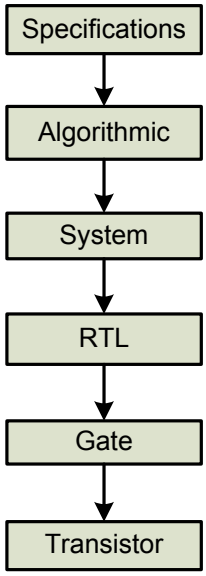

Figure 2–5: Design abstraction levels

to use or the number of abstraction levels that exist between the specifications and the final chip, we have decided to use three abstraction levels for this work based on verification requirements.

- Algorithmic level
- System Level

- RTL Level

According to [4], the verification effort can be reduced substantially through abstraction. Working at higher abstraction levels implies using less low-level implementation details. This results in faster testbench development and more manageable verification environment. However, high abstraction level designs are always refined down to the implementation level. The verification environment has to follow up in the same way. Instead of having independent verification platform, it may be necessary to navigate across abstraction levels. This thesis present a novel approach to this new verification problematic.

2.2.3 Simulation Modes

Different simulation modes describe different levels of a signal's physical accuracy:

- **Electrical mode**. Represents physical electrical values such as voltages, currents, and impedances as waveforms described by a set of real numbers.
- **Logic mode**. The physical values are represented by a logic abstraction of the corresponding state values. The most common discrete logic values are: O, 1, floating, and undefined. Floating is used in modeling high impedance states. The pure abstraction from electrical waveforms to logic states loses information necessary to accurately model hazards, illegal states, and races. There is a tradeoff between physical accuracy and simulation speed. Physical accuracy may also be traded for simulation complexity. Additional states are sometimes introduced to increase model accuracy, but they result in an increase in complexity and a decrease in speed.

2.2.4 Time Representation

At different levels of abstraction, time is generally represented either in terms of real numbers or integers:

- **Real numbers**. Are used to express time in electrical-mode simulation, which is measured in the same units as it is in the real world. For example, time can be expressed as hours, seconds, nanoseconds, or picoseconds. Real numbers make it possible to express small time steps that usually occur in integrated circuit simulations.
- **Integers**. Are used to represent time at higher levels of abstraction, in which time is generally an integer multiple of a basic unit of time. For example, gate delays and clock periods correspond to a multiple of the basic time unit.

2.2.5 Scheduling Policy

Simulation scheduling specifies how often elements are evaluated:

- Time driven. In the most direct electrical-mode simulation (direct methods), all the elements are simulated at every time-step. The step size could vary, but independently of the activity of the signals. The simulation in this case is driven by the time steps.
- Event driven. There is a notion of event in the simulation, a change in state of some node in the circuit. Only device elements that are affected by a node event are scheduled to be re-evaluated. The simulation in this case is driven by events. In the particular case where an event is not associated with a given time, the simulation sometimes is also referred as data-driven.

2.3 Co-Simulation Framework

As mentioned before, a primary objective of this research is to reduce the total effort put in simulation-based verification. Verification is a simulation-intensive task. As expected, simulation time grows as the square function of the system complexity [4]. Since there is a need for longer, larger and more realistic simulations performed within a finite time-to-market, complex VLSI systems chips currently require millions of hours of simulation time to validate and verify designs. Usually several simulators are used in the design pass [65] because no single simulator addresses all levels of abstraction, modeling, performance and verification issues during all stages of design. This includes early software development, system-level exploration and design, logic-level simulation, and circuit-level simulation. Another reason for the use of multiple simulators in the design stage is that models sometimes are only available in certain languages, and companies want to reuse this models, sharing them across company groups and even with other companies. To address this heterogeneous simulation reality, several commercial tools and co-simulation methods have been proposed. This section presents an extensive state-of-the art study of both co-simulation methods and commercial tools that are related to the proposed solution.

In the past decade, extensive work has been done in the field of co-simulation, especially in the field of simulation-based verification. The result is a plethora of co-simulation frameworks (e.g., [65, 58, 71, 22, 3]), coming from both academic and commercial worlds. To our best knowledge, none of the prior work addresses the multi-abstraction level verification problem and vertical testbench reuse for complex digital designs.

One approach of performing co-simulation is by adopting high-level languages such C/C++ and Java. When applying these techniques, the hardware-software co-simulation can be performed by compiling the designs using their high-level language compilers and running the executable files resulting from the compilation process. There are several commercial tools based on C/C++. Examples of such co-simulation techniques include Catapult C from [26] and Impulse C, which is used by the CoDeveloper design tool from [66]. In addition to supporting the standard ANSI/ISO C, both Catapult C and Impulse C provide language extensions for specifying hardware implementation properties. An application designer describes his/her designs using these extended C/C++ languages, compile the designs using standard C/C++ compilers, generate the binary executable files, and verify the functional behavior of the designs by analyzing the output of the executable files. To obtain the cycle-accurate functional behavior of the designs, the application designer still needs to generate the VHDL simulation models of the designs, and perform low-level simulation using cycle-accurate hardware simulators. The DK Design Suite tool from Celoxica supports system level development using Handel-C [1] and SystemC [34], extensions of C/C++ language. While Handel-C and SystemC allows for the description of hardware and software designs at different abstraction levels. However, to make a design described using Handel-C or SystemC suitable for direct register transfer level generation, the designer needs to write his/her designs at nearly the same level of abstraction as handcrafted register transfer level hardware implementations.

Cadence Design Systems [10] proposes the Incisive Functional Verification platform. Their unified verification methodology is centered on the creation and use

of a transaction-level "golden" representation of the design and a verification environment called the functional virtual prototype. Others, such as Mentor Graphics Scalable Verification Solution [18], propose tools that scale with design complexity and utilize multiple levels of abstraction. Synopsys on their side offers the Discovery Verification Platform [59] that is an integrated system, RTL, equivalence checking, and mixed-signal verification solution. Finally, Verisity Verification Process Automation (VPA) Systems [68] automates the entire verification process through the use of their proprietary language *e*.

All these tools support the SystemC language for the design process, but none of them actually use it as the central component for functional verification. As it will be presented in Chapter 3, SystemC is a powerful open source language that can be used efficiently as a backbone for the whole verification system. SystemC is abstraction level-independent, making vertical testbench reuse a reality. Furthermore, none of these commercial tools proposed to use Matlab and Simulink to assist the verification system. Each of these tools barely support third party software to retain their own proprietary control. These big companies are making really good tools, but we have to remember that the main objective behind all these products and unified solutions is the profit.

In the next chapters, we present new mechanisms that will be the foundation of a novel verification platform based on the open-source language SystemC. This new verification methodology aims at using the right tool for the right task by providing modular and flexible interfaces. The complexity of current digital systems

necessitates a variety of specialized tools, so it is important for a verification platform to provide the right mechanisms and methodology to be considered a successful verification solution.

Chapter 3
Multi-Abstraction Level Verification

This chapter describes a comprehensive infrastructure to efficiently address challenges faced by designers when trying to verify designs represented at multiple abstraction levels. As we saw in previous chapter, various challenges encountered in verification require new methodologies to solve the verification bottleneck. The next sections present a platform combined with a novel methodology that provides efficient mechanisms to considerably reduce the time spent on verification. The design flow targeted with this methodology will be presented first and then the verification challenges will be highlighted. The key element of this platform is the interface used by our verification system to bridge the gap between algorithmic modeling and lower abstraction levels.

3.1 The Design Flow

Increasing complexity of digital systems forces designers to move up across abstraction levels. They no longer start the design at the register transfer level using VHDL or Verilog languages. Many architectural issues, like hardware-software partitioning, need to be undertaken well before physical implementation can be done. As a result, system design starts from handwritten specifications and will go all the way through multiple abstraction refinement stages.

At a very high level, designers are concerned with the general architecture of the system and finding the optimal way to implement the specifications. While register

transfer level design tasks benefit from robust methodologies and a restricted choice of design languages, higher abstraction level activities have less limitation. Depending mainly on the application domain, a large variety of languages and tools are used to explore the design space. In this work, we address the specific application domain of digital design. The conception of these kinds of systems usually begin with tools like MATLAB and Simulink to create and optimize the algorithm in order to meet specific performances. Figure 3–1 depicts a typical design flow where the modeling stages are separated into three abstraction levels: algorithmic level, system-level, and register transfer level. This design flow shows a top-down approach that has been simplified to reveal only the key tasks that need to be done in order to design, validate, and verify at each abstraction level. The reality of an industrial development may be more complex, involving more steps and many iterations through various portions of this flow, until the final design converges to a form that meets the specification requirements of functionality, area, timing, power, and cost. The following sections will present, in more detail, each of these modeling phases and how they will affect the overall system validation and verification.

3.1.1 Algorithmic Level Modeling

The first modeling step in the design flow, as shown in Figure 3–1, is to create an algorithmic model of the system to explore different ways of processing the input data. Designers are given the system specifications; a document describing a set of functionalities that the final solution will have to provide and a set of constraints that it must satisfy. In this context, the algorithmic model development is the initial process of deriving a potential computational procedure for solving the problem

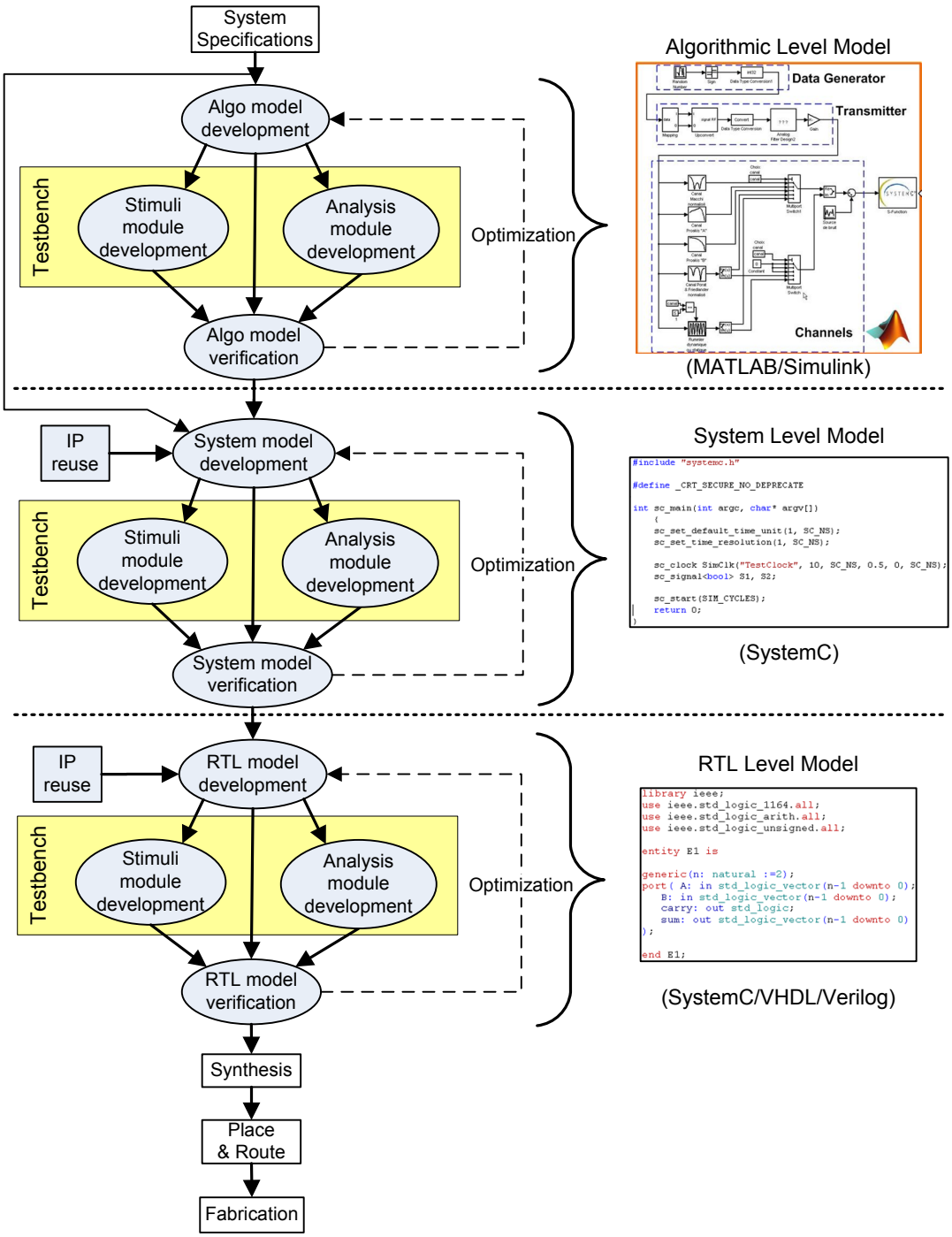

Figure 3–1: Design Flow

resulting from the design specifications and requirements. At this level of abstraction, the system is represented with individual processing elements that constitute the algorithm. There is no distinction between what will be the hardware component or the software program since no partitioning has been done yet.

MATLAB [38], a product of Mathworks Inc., is a popular programming language and development environment for numeric applications. The MATLAB product family provides a high-level programming language and an interactive technical computing environment perfectly adapted to our algorithmic level modeling needs. It includes a variety of functions for algorithm development, data analysis and visualization, and numeric computation. The main strengths of MATLAB lie both in its interactive nature, which makes it a handy exploration tool, and the richness of its precompiled libraries and toolboxes, which can be combined to solve complex problems.

MATLAB is further enhanced with Simulink; a platform for multi-domain simulation and model-based design for dynamic systems. It provides an interactive graphical environment and a customizable set of block libraries that can be extended for specialized applications. Simulink is integrated with MATLAB, providing immediate access to an extensive range of tools.

Designers of complex numerical applications commonly use MATLAB and Simulink as a preliminary modeling medium to quickly experiment with different algorithms that will best represent the system [7, 54]. Using the convivial graphical user interface, the system is built by dragging and dropping blocks from the library browser onto the graphical editor and connecting them with lines that establish mathematical

relationships between the blocks. The result of this modeling phase is an easy way to manipulate the graphical model of the system that can be dynamically simulated to prove or refute the validity of the initial concept. The model is also used to fine-tune various parameters or to help find potentially intensive computing blocks in the system. Once the algorithmic model has been extensively simulated and is considered satisfactory, the design is then further refined to the next abstraction level in the design flow.

3.1.2 System-Level Modeling

The design team proceeds to the system-level modeling phase using the algorithm and simulation results, previously created with MATLAB and Simulink, and the system specifications. During this phase, all aspects of the system are modeled using a hierarchal approach, so that a single designer can concentrate on a portion of the model at any given time. Thus, the architectural description provides a partition of the design in distinct modules, each of which contributes a specific functionality to the overall design. These modules have well defined input/output interfaces and protocols for communicating with the other components of the design. In order to save development time and cost, some of these modules come from a library of pre-design blocks. This technique is referred to as Intellectual Property (IP) reuse.

Recently, the industry has put emphasis on the importance of Electronic System-Level (ESL) [14] design tools and methodologies in the development of complex devices like system-on-chip (SoC). A broad range of ESL design tools have emerged on the market in the past few years from a variety of companies like Celoxica, CoWare,

Summit, Cadence, Mentor Graphics, Synopsys etc., to name only a few. This work uses SystemC for system level modeling.

SystemC is an open-source system-level modeling language based on C++. It provides hardware-oriented constructs as a class library implemented in standard C++. The SystemC design flow starts from a highly abstract algorithmic system description and applies an iterative refinement process. Details regarding algorithm partitioning, timing, process scheduling, data representation, hardware and software partitioning are progressively added. The multi-level abstraction design methodology is one of the most important properties of SystemC. To support modeling at different levels of abstraction, from the system-level to the component level, SystemC supports a rich set of port and signal types. Using this approach, system-level designers can create detailed descriptions and perform faster cycle-accurate simulations, when compared to an equivalent RTL simulation. The gain in simulation time approaches an order of magnitude [11].

The result from this design phase is an executable specification of the system that simulates the behavior of the design. It is used for architectural exploration and performance analysis. Critical issues like hardware/software tradeoffs and interfaces optimization are investigated with this model. After extensive validation and verification, both hardware and software development can ramp up simultaneously. The system-level model becomes both the "golden" reference design and the prototype. The golden reference design drives the hardware development. Meanwhile, embedded software is developed on the prototype. Using the SystemC language, hardware designers refine the system-level model to the register transfer level.

3.1.3 Register Transfer Level Modeling

From the refined system level model, the hardware design team proceeds to the Register Transfer Level (RTL) design phase. Hardware Design Languages (HDL), like VHDL and Verilog, are used to design each functional component of the system-level model. To save time manually translating SystemC to HDL, a subset of SystemC allows designers to use it as an HDL. Some SystemC compilers have begun to emerge on the market like the Celoxica Agility Compiler that provide behavioral design and synthesis for SystemC. This phase also sees the development of the clocking system of the design and architectural trade-offs such as speed/power.

These HDL descriptions are then successively transformed into block-level, gate-level, transistor-level and finally into mask-level layout for fabrication. Computer Aided Design (CAD) software tools such as synthesis and place/route are extensively used throughout this design process. Most of the activities are semi-automatic or at least heavily supported by CAD tools.

3.2 Design Verification

As discussed in the previous chapter, the correctness of a digital circuit is a major consideration in the design of digital systems. Manufacturing microchips involves extremely high and increasing costs, so the consequences of flaws going unnoticed in system designs until after production phase are very expensive. At the same time, verification of digital systems is still one of the most challenging activities in chip development [70]. As of today, verification is still carried out mostly with ad-hoc test, scripts and tools developed by the design and verification teams specifically

for the current design [47]. In the best case, these verification infrastructure developments can be amortized among a family of designs with similar architecture and functionality [55]. Moreover, verification methodology still lacks any standard or even a commonly accepted approach, with the consequence that each hardware engineering team has its own distinct verification practices which often change with subsequent designs by the same team, due to the insufficient "correctness confidence level" that any of the current approaches provide. Given this scenario, it is easy to see why many digital integrated circuit (IC) development teams report that around 70% of the design time and engineering resources are spent in verification, and why verification is thus the bottleneck in the time-to-market for IC development. In this section, we will examine the verification tasks associated with the design flow presented in the previous section. Time consuming steps will be highlighted and a novel solution will be presented.

3.2.1 Testbench Creation

At each abstraction level in the design flow, extensive verification of the model needs to be done in order to eliminate design flaws as soon as possible. Errors related to system architecture and functionality are easier and less expensive to find early in the design flow. As more and more abstraction details are added to the model, it becomes difficult and time-consuming to find them. As presented in the previous chapter, one way to verify our model is through simulation; and simulation implies testbench. Figure 3–2 shows the basic environment required in order to verify a model using simulation. We will refer to this model as the Design Under Verification (DUV). A testbench is created around the DUV to send stimuli to the

inputs and analyze the response at the outputs. The stimulus generator portion of the verification environment is responsible for creating the right stimuli required to completely exercise the design. All possible behaviors of the DUV should be demonstrated during the simulation. To confirm that these behaviors have been exercised and to verify that they conform to the device specification, a response checking module is connected to the outputs of the DUV. The testbench emulates the system environment of the design. Depending on the complexity of the DUV, the testbench consists of different test techniques to efficiently verify the design. As an example, random test generation is often used in the stimuli generator for a DUV having lots of system states. Reference model comparison is a technique used to compare output responses of the DUV against a "golden" reference model. The challenge of testbench creation is how to offer efficient stimuli that can achieve high coverage of a design's function and self-checking mechanism to compare outputs with what is expected. A comprehensive testbench can, in fact, be more complex and lengthy (and take longer to develop) than the synthesizable circuit being tested [46].

The design flow of figure 3–1 clearly shows that the model of the design needs to be verified at each abstraction level to ensure that it behaves as expected. As a result, a testbench is also required at each abstraction level to simulate the model. Figure 3–3 recalls the testbench creation steps from the design flow presented in the previous section. The two most important tasks are the stimuli module development and the analysis module development.

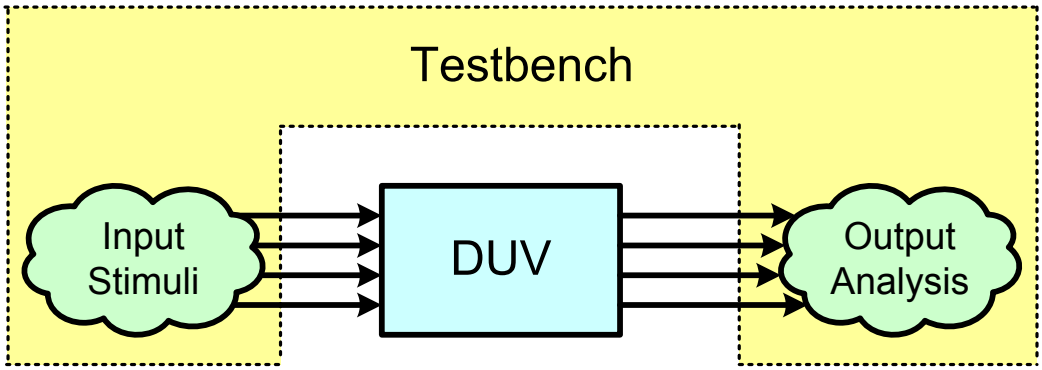

Figure 3–2: Model Simulation

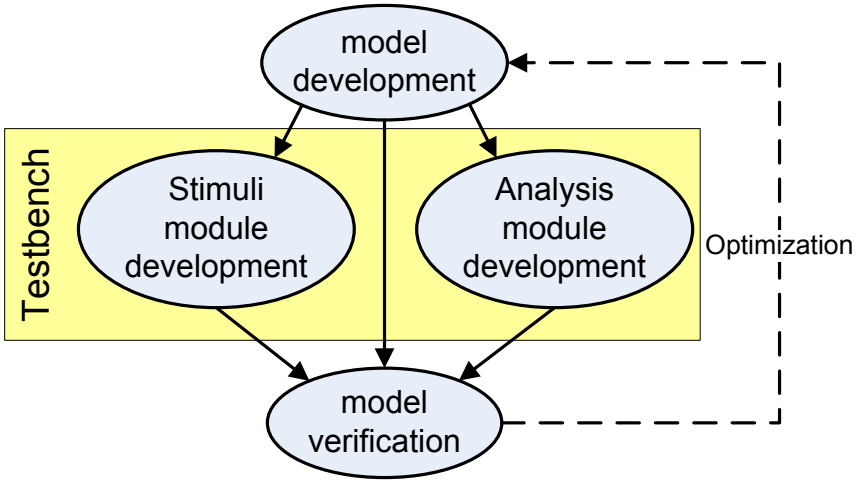

Figure 3–3: Testbench Creation

The main difficulty when working with a multi-abstraction design flow, like the one shown in Figure 3–1, is that each abstraction level uses its own language and software tool. Therefore, when it comes to verification, a completely new testbench, consisting of a stimuli module and an analysis module, needs to be created with the abstraction level language and software. There are two major problems related to this situation:

- Wrong software tool used for verification
- Testbench duplication across abstraction levels

The consequences are an inefficient testbench and time-consuming verification environment development. Using the design tool to verify the model of the system has proven to have many limitations for complex designs. As discussed in the previous chapter, hardware verification has undergone multiple transformations in the past years. One of them is using hardware verification languages (HVL) instead of hardware design language (HDL) to address complex testbench programming issues. Multi-abstraction level verification also needs to address this issue. The verification environment needs to be extracted from the design environment so it can tackle the verification problem with appropriate methodologies and verification techniques. Moreover, creating a new testbench at each abstraction level is a time-consuming task that results in duplicated verification components. Since each level deals with the same system, described in more or less detail, some elements in the testbench can be reused across the abstraction levels. For example, data generated at the algorithmic level, to verify that the algorithm behaves as specified, is also required at the system-level or register-transfer level to verify the hardware implementation of the

algorithm. The next section will demonstrate how we tackle these problems using the SystemC modeling language.

3.2.2 SystemC Verification Platform

In section 3.1.2, we presented SystemC as a system-level design language based on C++. Recently, a working group within the Open SystemC Initiative (OSCI) released the SystemC Verification (SCV) standard that extends the language to verification. This working group includes a number of EDA companies, semiconductor developers, and system/IP companies. It includes ARM, Cadence Design Systems, CoWare, Forte, Fujitsu, Mentor Graphics, Motorola, ST Microelectronics, and Synopsys. In addition, representatives from a variety of academic institutions made key contributions. These individuals performed extensive research in SystemC and verification. Among the institutions represented were the University of Chemnitz and the University of Tuebingen, both located in Germany.

Originally, the developers of SystemC provided a platform upon which developers could build various design methodologies. This working group has achieved a similar result for verification. Its goal was to define a set of verification classes within SystemC. These classes would provide a basis for developing various verification methodologies. This work uses the SystemC Verification Library v1.0 that is composed of the following features:

Data introspection: manipulation of arbitrary data types.

Randomization: generation of random values through the scv_random class that support advanced seed management and generation algorithm selection.

Constraints for randomization: creation of constraint expressions, with the scv_constraint_base class, to specify the range of legal values.

Weight for randomization: possibility to bias the random values generation process, with the scv_bag class, so that some values are generated more often than others.

Transaction-based verification: modeling style for test bench with transactors and transaction recording through scv_tr_db, scv_tr_stream, and scv_tr_stream.

A verification methodology can be build using these key capabilities within SystemC [45, 64, 17]. Our platform uses SystemC as the backbone for the verification environment. Figure 3–4 presents an improved version of the design flow using our novel verification platform. The key characteristic of this platform is its ability to verify the design at multiple abstraction levels. Looking at figure 3–4, we see how the verification environment spreads out vertically and overlaps abstraction levels boundaries. There are three main advantages of this layout. First, verification begins earlier in the development cycle. With design complexity rapidly increasing, designers need to verify performance as early in the lifecycle as possible. Verifying performance as the product transitions between each phase reduces risk by finding critical issues early in simulation, instead of later on in the product development cycle, where it is more time-consuming and costly to fix. The next advantage is to use the same language for verification all along in the design flow. It is easier to make verification components communicate together when they are written in the same

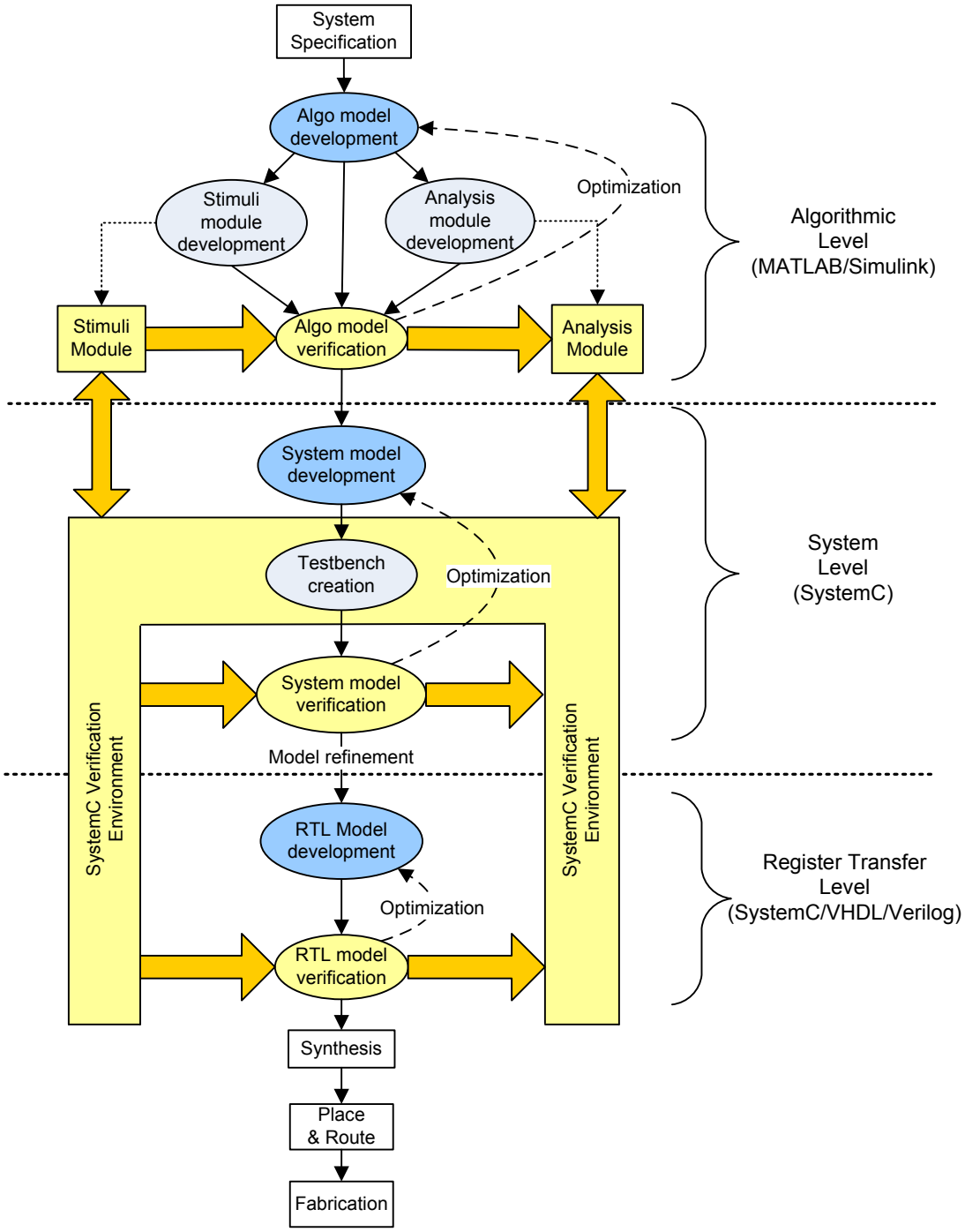

Figure 3–4: Improved Design Flow

language. Using SystemC as the central element of the verification environment ensures uniformity among the testbench. Also, having only one language to learn helps verification engineers to maintain testbenches. The third benefit, where lots of time can be saved, is the reuse of verification components. In the typical design flow presented in the previous section, testbenches tends to be separated by the abstraction levels having almost no interaction between them. In this context, reuse is almost impossible because each testbench is written explicitly for a particular abstraction level, using different languages. Our platform provides the right mechanisms such that all abstraction level verification components can communicate together to share their specific resources. The central testbench in SystemC reuses the verification elements scattered at each abstraction level. As an example, a data generator modeled in MATLAB/Simulink can be reused by the SystemC verification environment to produce relevant stimuli to the DUV described in HDL. Data exchange between SystemC and MATLAB/Simulink is possible through our novel co-simulation interface SimSyC. The next section presents in detail the structure of this interface and how it can be used to considerably reduce the time spent on model verification.

3.3 The SimSyc Cosimulation Interface

The first discontinuity in the verification flow happens when the design is taken from the algorithmic level to the system level. As a result of the programming paradigm shift, traditional verification is performed independently with two distinctive testbenches. As presented in the previous section, this approach is error prone and time consuming. In an effort to bridge the verification gap that exists between these abstraction levels, we created a novel cosimulation interface between Simulink

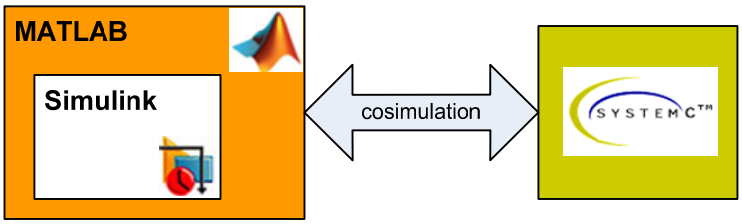

Figure 3–5: Simulink and SystemC Cosimulation

and SystemC called SimSyC. In this section, we present the underlying details of SimSyC and how it is used to improve the verification task. The goal of the interface is to cosimulate a Simulink model with a SystemC model. Figure 3–5 represents the cosimulation connection between both software tools. Simulink is embedded in the MATLAB software. Therefore, SimSyC will first establish a connection between SystemC and MATLAB and then get access to Simulink. SimSyC's main task is to exchange data between Simulink and SystemC and to synchronize both simulation kernels. There are multiple application scenarios for this interface that will be discussed in the following section.

3.3.1 Typical Applications

The SimSyC cosimulation interface can be used in different ways. There are four primary application scenarios that were identified and illustrated in figure 3–6. All these scenarios were elaborated with one objective in mind: to assist the designer in his verification efforts. Therefore, the design to verify is the central component of each of these scenarios. Figure 3–6 (a) shows that SimSyC can be used to verify a SystemC design with a model of the system environment running in Simulink. This configuration helps the verification task in two different ways. First, the time spent on testbench creation is considerably reduced by using the interactive graphical

environment of Simulink and the variety of a customizable set of block libraries. A complex model of the system environment can be created in a few hours by using predefined blocks. Moreover, the quality of the testbench is increased because the Simulink blocks used have been already verified. Therefore, the testbench creation step is less prone to errors and provides a robust model to stimulate the design under verification. In the second scenario of Figure 3–6 (b) Simulink is used as a graphical display to enhance SystemC simulation. Stand alone SystemC simulator only offers text based output. Limited verification and debugging can be done in this context. SimSyC opens up to a wide range of graphical tools that can be used to display SystemC output data directly in Simulink. For example a SystemC digital signal processing design can use the discrete-time eye diagram scope found in the communication blockset of Simulink to plot a modulated signal. This eye diagram can be used to reveal the modulation characteristics of the signal, such as pulse shaping or channel distortions.

The third configuration shown in Figure 3–6 (c) is used to simulate a design fragmented across two different abstraction levels. One part of the design is a SystemC model while the other part is a Simulink model. The advantage of this layout is to start the verification of the SystemC section of the design earlier. Even if the other part is still in Simulink and not yet available in SystemC, this provides continuous verification during the transition from a higher abstraction level to a lower one.

Figure 3–6 (d) shows the last application scenario where two versions of the same design are simulated together. The first version (Design v1) is an algorithmic model in Simulink. The second one (Design v2) is a SystemC model obtained through the

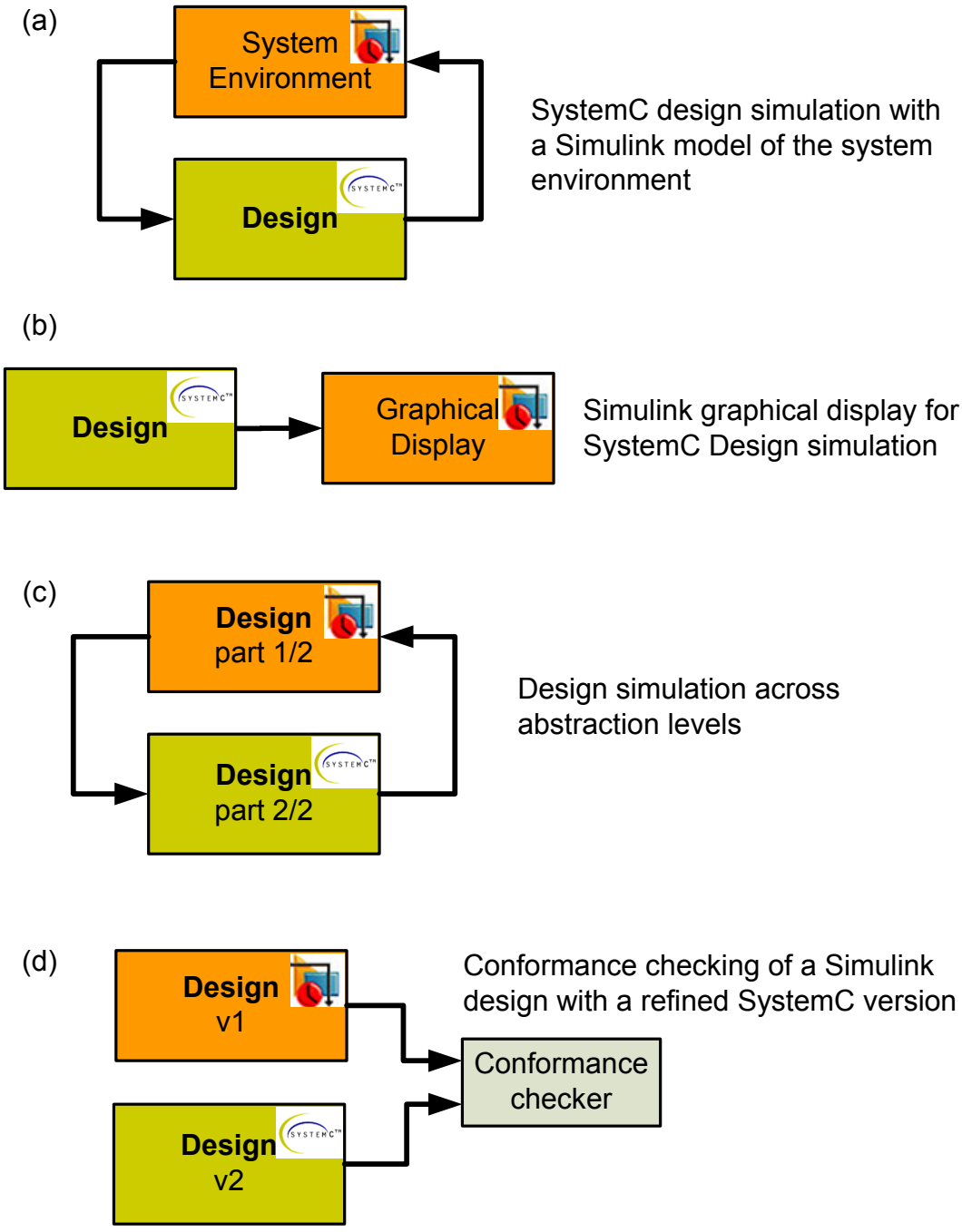

Figure 3–6: Application scenarios

refinement process. The outputs of both designs are collected to perform conformance checking. For instance, the Simulink design can represent a golden model to verify a lower level SystemC implementation. Important information is collected during this verification step to insure that the transformation from Simulink to SystemC has been done properly. This section has only presented the four application scenarios where SimSyC can be used. More examples and experimental results will be presented in chapter 5.

3.3.2 Related work

Connecting Simulink and SystemC together has already been tried in the literature. Authors in [12] propose a solution to integrate SystemC models in Simulink. A wrapper is created using S-Functions to combine SystemC modules with Simulink. This wrapper initializes the SystemC kernel and converts Simulink data type to SystemC signals and vice versa. Simulation control is entirely handled by Simulink. Some extensions of the SystemC kernel are required for initialization and simulation tasks. In [69], SystemC calls MATLAB using the engine library. MATLAB provides interfaces to external routines written in other programming languages. Using the C engine library, it is possible to share data between SystemC models and MATLAB. This simple working demo shows how to use the library to send and retrieve data from the MATLAB workspace and plot some results. The main difference with [12] is with the simulation control: SystemC is now the master of the simulation and MATLAB operates as a slave process. Also, Simulink is not supported in this example. In a similar way, MathWorks provides a commercial solution to close the gap between algorithmic domain and the hardware design. The link for ModelSim [39] is

a cosimulation interface that integrates MATLAB and Simulink into the hardware design flow. It provides a link between MATLAB/Simulink and Model Technology's HDL simulator, ModelSim. This interface makes the verification and cosimulation of RTL-level models possible from within MATLAB and Simulink. As opposed to the two previous techniques, there is no support for system level languages like SystemC. These approaches [69, 12, 39] all try to reduce the barrier that exist between higher level modeling and existing hardware design flow. While [39] is a fully functional commercial tool for RTL verification, [69, 12] suffer from their embryonic stage (i.e. incomplete solutions for hardware design and verification).

The authors in [8] have look at the problem of cosimulating continuous systems with discrete systems. The increasing complexity of continuous/discrete systems makes their simulation and validation a demanding task for the design of heterogeneous systems. They propose a cosimulation interface approach based on Simulink and SystemC. The main objective of the proposed solution is to provide a framework to evaluate continuous/discrete systems modeling and simulation.

SimSyC tries to push the idea a step further than just a cosimulation interface; it is a complete verification solution. It uses MATLAB external interfaces, similar to the example described in [12], to exchange data between SystemC and Simulink. Once this link is established, it opens up a wide range of additional capability to SystemC, like stimulus generation and data visualization. The first advantage of our technique is to use the right tool for the right task. Complex stimulus generation and signal processing visualization are carried out with MATLAB and Simulink while hardware verification is performed with SystemC verification standard. The second

advantage is to have a SystemC centric approach allowing greater flexibility and configurability.

3.3.3 Master-Slave Relationship

All the scenarios presented in section 3.3.1 show how Simulink and SystemC interact together. Both simulation kernels can send and receive data through the SimSyC interface. However, to insure proper coordination of the cosimulation, we must define who will be the master and who will be the slave. The master will have two additional tasks to take care of beside design simulation. It must first initialize the different components of the system before starting the simulation. Then, it must continually control and monitor simulation execution. As an example, MathWorks' link for Modelsim [39] presented in chapter 1 uses Simulink as the master and ModelSim as the slave. This configuration is Simulink centric and thus vendor dependent. Initialization and simulation control is performed by Simulink over the foreign language interface (FLI) of ModelSim. SimSyC takes a different approach. SystemC is the master while Simulink is the slave. This topology has the advantage of being vendor independent. The core of the cosimulation interface is written using the open source language SystemC.

3.3.4 Modes of Communication

Communication between Simulink and SystemC can be done in various ways. Three options were considered for this interface. Data transfer can be done through hard disk files, TCP/IP Ethernet connection and shared memory. To choose the optimal mode of communication, several criteria were analyzed.

First, data transfer between Simulink and SystemC must be fast enough to sustain the amount of information to be exchanged. There are basically two categories of information that transit through this link. First, control data assures proper simulator initialization and synchronization. The bandwidth required for these tasks is relatively low. Initialization commands are used off line, having no direct impact on the simulation run time. Regarding synchronization, a simple handshake protocol has been defined to minimize the interface usage. Section 3.4.3 gives more details about how simulator synchronization has been implemented. The second type of information that travels over the link is the actual simulation data. Transfer speed is now an important issue if we want to minimize the impact on simulation runtime. According to the typical applications presented in section 3.3.1, simulation data comes from either Simulink data generator models or SystemC data analysis modules. Depending on the design application field and complexity, the simulation data can be of various sizes and have more or less real time requirements. For example, consider a data generator, used to verify an image processing design, which sends a burst of data when required by the design. This data transfer does not occur on each simulation clock, thus have minimal real time requirements. On the other hand, data processing intensive designs, like telecommunication signal processing, may require a continuous real time data flow from the generator. This will have a significant impact on simulation time if the interface between MATLAB and SystemC is not fast enough. Since the transfer speed is closely related to the design application, the fastest communication mode will be preferred to support a wider range of application.

The second criterion considered for the communication mode was flexibility. As stated previously, the application that will use this cosimulation link is unknown at this point. Therefore, the communication mode of the interface must be flexible enough to be customized for application specific requirements. For example, the number of input and output ports and their corresponding data type are variable from one design to another. Consequently, inherent flexibility within the communication mode is an important issue if we want to minimize software processing overhead converting data types or manipulating data structures.

The last decisive factor taken into consideration was the potential of integration within MATLAB and SystemC. To be efficient, the co-simulation link has to be easily connected to MATLAB and SystemC without any major modification to both environments. Moreover, co-simulating these two programs is only possible with a good hand-shake mechanism. The communication mode must be able to access simulation parameters to control data exchanges.

Table 3–1 below summarizes the main characteristics of each communication mode regrouped according to the decision criteria just presented. On the second line we have the hard disk approach which consists of transferring data between MATLAB and SystemC using a file system that resides on the hard disk. This technique suffers seriously from its poor transfer speed and lack of flexibility. Access to the computer hard disk is definitively a slow operation that will have a big impact on long real time simulation runs. Also, data organization into multiple files will have to be managed manually; adding unwanted development time and complexity to the interface. On the other hand, limited simulation control can be achieved using the file

Communication modes	Speed	Flexibility	Integration
Hard disk	Read and write to hard disk is a relatively slow operation	Limited control through files - Complex data organization - Large files to manipulate	Native file access support in both MATLAB and SystemC
TCP/IP (Ethernet)	- Speed depends on network congestion - Requires an efficient protocol	- Protocol dependent - Requires data types conversion functions - Multiple computers	Integration to MATLAB and SystemC using WinSock library
Shared Memory	- Fast memory access - No protocol needed	- Unlimited flexibility through memory allocation - Native data types conversion functions - Data organization simplify with MATLAB workspace - Direct MATLAB configuration	Native support in both MATLAB and SystemC through the Engine library

Table 3–1: Main characteristics of communication modes

system. Access to simulation parameters will have to be managed by additional subroutines, thus adding complexity and processing overhead to the interface. Finally, both MATLAB and SystemC include native support for file access.

The next method presented in table 3–1 uses an Ethernet connection to exchange data between MATLAB and SystemC using TCP/IP. The main advantage of this approach is that MATLAB and SystemC can run on two different computers. This may be a valuable feature for large designs with complex testbenches that involve very long simulation runs. However, transferring data over an Ethernet link requires

an efficient protocol. Simulation data must be converted into network packets and a well organized hand-shake must be implemented. Transfer speed will depend on the efficiency of this protocol and the network congestion of the link. Simulation control over TCP/IP can be done in a more efficient way than with the hard disk method. Integration to MATLAB and SystemC can be done using windows sockets through the WinSock library.

The last method showed in Table 3–1 uses shared memory. This communication mode offers the fastest data transfer speed using direct memory access. No protocol is required since data is written directly in software's local variables. Data transfer between MATLAB and SystemC is thus considerably simplified. MATLAB includes an application programming interface (API) to communicate with external third party software. This API is rich in features and provides a solid infrastructure to build on. The only restriction of shared memory is that both MATLAB and SystemC must be executed on the same host computer. This will result in simulation execution speed limitations for larger designs. Computing intensive designs and testbenches should use TCP/IP communication mode instead of shared memory to take advantage of distributed simulation execution. SimSyC assumes that both MATLAB and SystemC can run on the same computer to obtain satisfactory results. As a result, we chose shared memory to implement the cosimulation interface.

3.4 SimSyC Implementation Details

The SimSyC interface is made of two separated programs. One of them is written using the SystemC language and the other is a MATLAB C-MEX S-function. These two programs exchange data through the *engine* interface and MATLAB's

workspace. Figure 3–7 shows all the elements that are part of the SimSyC interface. First, a SystemC module (sc_module) uses the *engine* C++ library to manage all the initialization and control tasks. It will also ensure the synchronization between both simulators. Then we have the MATLAB command window where SystemC can execute MATLAB commands through the engine interface. SystemC now has access to the workspace of MATLAB; where all variables are stored. Using read/write commands, SystemC can freely read and write data to this workspace. The last element is Simulink, where the data generator/analysis model is simulated. A C-MEX S-Function block is added to the Simulink model to exchange data with MATLAB workspace. This block is written using the C++ language and is configurable according to the design requirements.

3.4.1 MATLAB Engine Library

Shared memory communication mode between MATLAB/Simulink and SystemC will be implemented using the MATLAB C language external interface. The *engine* library contains several routines to allow a C program to call MATLAB. Table 3–2 summarizes the routines available in this library. Using these routines, it is possible to control the MATLAB computation engine from a C program. On Microsoft Windows, the engine library communicates with MATLAB using a Component Object Model (COM) interface (UNIX uses pipes). The SystemC module employs these routines to remotely control MATLAB and exchange data back and forth between SystemC and the MATLAB workspace.

The MATLAB language works with only a single object type: MATLAB array. These arrays are manipulated in SystemC using the *mx* prefixed application

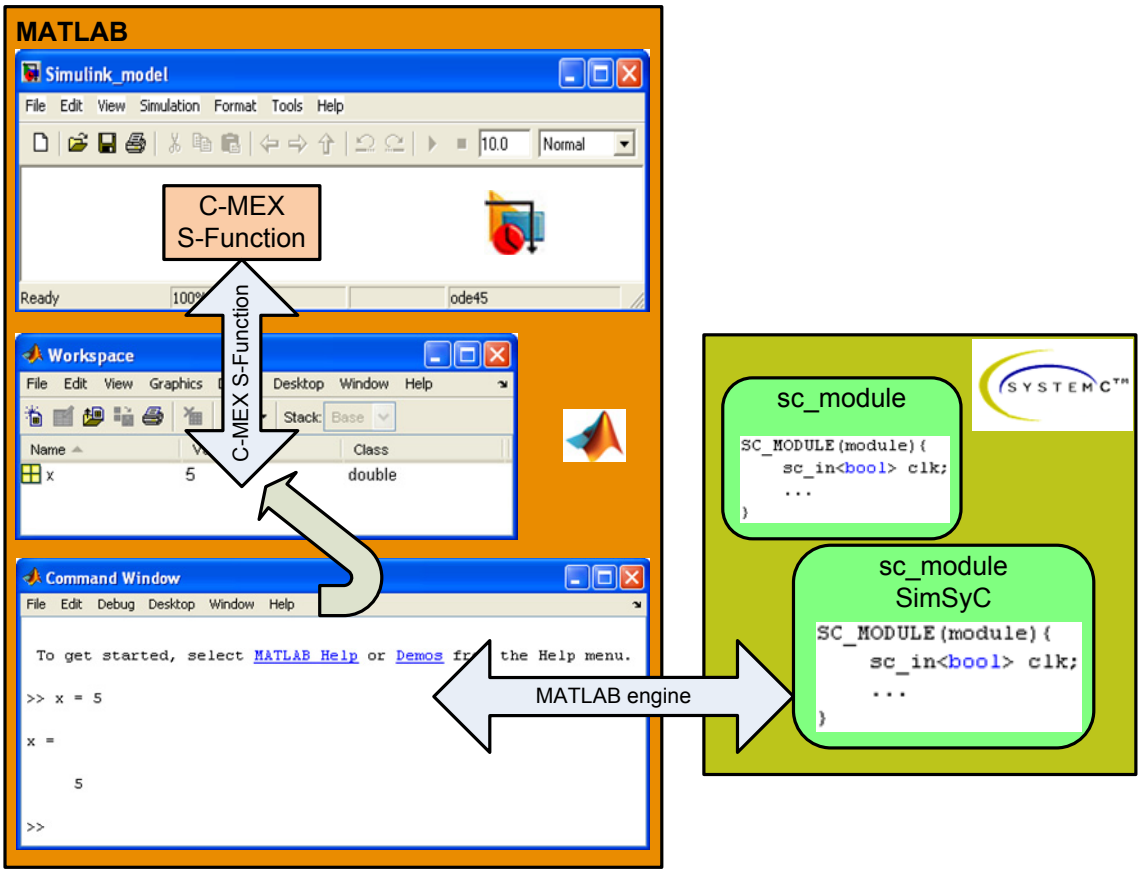

Figure 3–7: SimSyC internal components

Function	Description
engOpen	Start MATLAB engine
engClose	Shutdown MATLAB engine
engGetVariable	Get a MATLAB array from the engine
engPutVariable	Send a MATLAB array to the engine
engEvalString	Execute a MATLAB command
engOutputBuffer	Create a buffer to store MATLAB text output
engOpenSingleUse	Start MATLAB engine for non shared use
engGetVisible	Determine the visibility of MATLAB engine session
engSetVisible	Show or hide the MATLAB engine session

Table 3–2: MATLAB engine library routines

programming interface (API) routines included in the MATLAB engine. This API consists of over 60 routines to create, access, manipulate, and destroy mxArrays.

3.4.2 Simulink C-MEX S-Function

An S-Function is a program description of a Simulink block. S-functions can be written in MATLAB, C, C++ or Fortran, and are compiled as MATLAB EXecutable (MEX) files using the mex utility. A C-MEX program can be compiled to a dynamically linked library (DLL) and linked during simulation. Simulink interacts with a MEX file S-Function by invoking callback methods that the S-Function implements. Callback methods performed tasks required at each simulation stage. During the simulation of a model, Simulink calls the appropriate callback methods at each simulation stage. Figure 3–8 shows the main callback methods involve in a typical simulation run. All callback methods names have the *mdl* prefix.

Each method performs a predefined task, such as computing block outputs, required to simulate the block whose functionality the S-Function defines. M-files (written in MATLAB) have the advantage of avoiding the time-consuming compile-link-execute cycle required by development in a compiled language. However, the set of callback methods that C-MEX files can implement is much larger than the ones available for M-file S-Functions. Hence, C-MEX files can implement a wider set of block features such as the ability to handle matrix signals, complex inputs and multiples data types (fixed point data types). Moreover, the compiled C-MEX program can be dynamically linked with SystemC. That is the main reason we wrote SimSyC using a C-MEX S-Function.

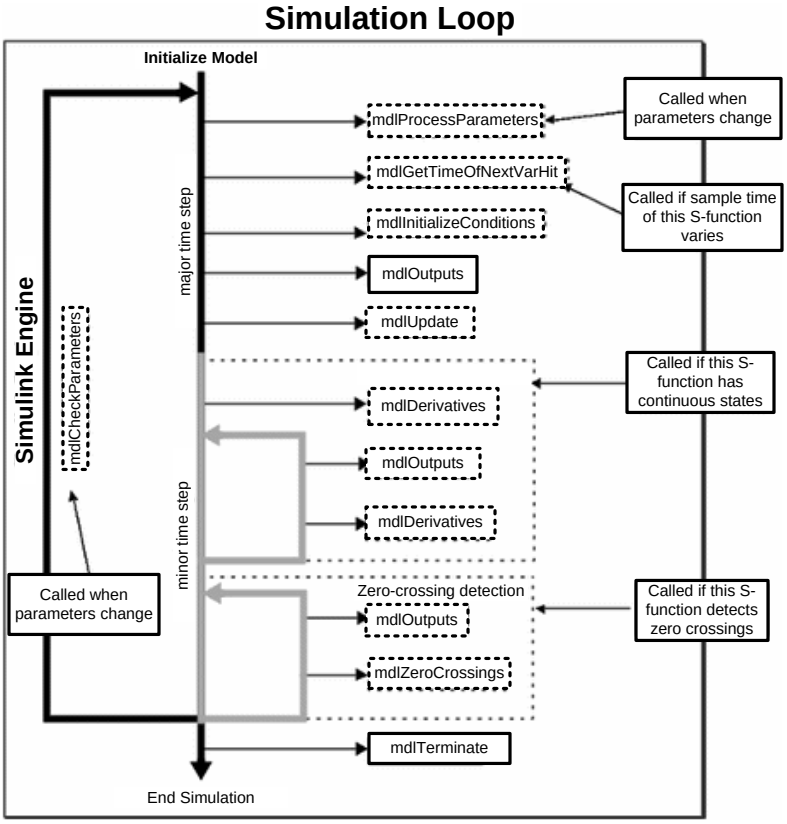

Figure 3–8: Simulink simulation stages (©1994-2006 The MathWorks, Inc.)

The easiest way to create a C-MEX S-Function is to use the S-Function Builder which is located in the Simulink Function and Table library. This eliminates the need to write an S-Function from scratch. However, its functionality is limited in the kinds of S-Functions that it can produce. For example, the S-Function Builder is limited to S-Functions that have no more than one input or one output. Also, data type is limited to non complex input or output signals and double precision signals manipulation. It is preferable to use skeleton implementations of callback methods,

called templates. The templates contain skeleton implementations of callback methods. The file *matlabroot/simulink/scr/sfuntmpl_doc* provides a detailed description of a C-MEX S-Function template. A general format of a C-MEX S-Function is shown below:

```
#define S_FUNCTION_NAME  your_sfunction_name_here
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"

static void mdlInitializeSizes(SimStruct *S)
{
}/* end mdlOutputs */
static void mdlOutputs(SimStruct *S, int_T tid)
{
} /* end mdlOutputs */

static void mdlTerminate(SimStruct *S)
{
}

/*=============================*
 * Required S-function trailer *
 *=============================*/
```

```
#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

The mdlnitializeSizes(SimStruct *S) method should specify the dimensionality, the data type, and the numeric type of each input and output port. It is also used to handle the parameters that are passed to the S-function through the dialog box. It accesses the parameters using the ssGetSFcnParam macro. We now describe how a programmer can specify the dimensionality , data type, and numeric type of each input/output port.

DIMENSIONALITY

The following methods exist for setting the input port dimensions:

- If the input signal is one-dimensional, and the input port width is w, the following method sets the dimension of the input port with index inputPortIdx:

  `ssSetInputPortVectorDimension(S, inputPortIdx, w)`

- If the input signal is a matrix of dimension m x n, then use the following method:

  `ssSetInputPortMatrixDimensions(S, inputPortIdx, m,n)`

If the S-Function does not require that an input signal has a specific dimensionality, we could set the dimensionality of the input port to match the dimensionality of the signal actually connected to the port. The following choices are available:

- If the input port can accept a signal of any dimensionality, the following method sets the dimensionality of the input port to the dimensionality of the signal connected to that port:

  `ssSetInputPortDimensionInfo(S, inputPortIdx, DYNAMIC_DIMENSION)`

- If the input port can only accept vector (1-D) signals but signals can be of any size, invoke the following method:

  `ssSetInputPortWidth(S, inputPortIdx, DYNAMICALLY_SIZED)`

- If the input port can only accept matrix signals but can accept any row or column size, invoke this method:

  `ssSetInputPortMatrixDimensions(S, inputPortIdx, m,n)`

  `where m and n are DYNAMICALLY_SIZE`

The programmer should also provide a **mdlSetInputPortDimensionInfo** method that sets the dimensions of the input port to the size of the signal connected to the input port. Simulink invokes this method during signal propagation when it has determined the dimensionality of the signal connected to the input port.

Finally, we have to provide a **mdlSetDefaultPortDimensionInfo** method that sets the dimensions to a default value, when Simulink cannot determine the dimensionality of the signal connected to some or all of the block's input ports. The

same macros (ssSetOutput(...), instead of ssSetInput(...)) are used to set the dimensionality of each output ports.

DATA TYPE

In order to set the data type of an input port we should invoke the method ssGetInputPortDataType(S, inputPortIdx, DTypeId id). The enumerated type DTypeId, is defined in simstruct.h file. The following table shows the equivalency of some Simulink, MATLAB, and C data types.

Simulink Data Type DtypeId	**MATLAB Data Type**	**C-Data Type**
SS_DOUBLE	mxDOUBLE_CLASS	real_T
SS_INT8	mxINT8_CLASS	int8_T
SS_INT16	mxINT16_CLASS	int16_T
SS_UINT16	mxUINT16_CLASS	uint16_T

Table 3–3: Data types equivalency

If the input data type is inherited from the block connected to the port, set the data type to DYNAMICALLY_TYPED. The data type of an input port is double (real_T) by default.

NUMERIC TYPE

The method **mdlInitializeSizes** is also used to set the numeric type of each input and output port. The following options are available:

- If the numeric type of the input port is real then invoke the method

  `ssSetInputPortComplexSignal(S, inputPortIdx, COMPLEX_NO)`

- If the numeric type of the input port is complex then invoke the method

`ssSetInputPortComplexSignal(S, inputPortIdx, COMPLEX_YES)`

- If the numeric type of the input port is ingerited from the block it is connected to, use the method

  `ssSetInputportComplexSignal(S, inputPortIdx, COMPLEX_INHERITED)`

The purpose of the **mdlOutputs** is to compute the signals that the block emits. Simulink invokes this required method at each simulation time step. The method should compute the S-Function's outputs at the current time step and store the results in the S-function's output signal arrays. Inside the **mdlOutputs** method the S-Function accesses the signals appearing in the input ports. An S-Function accesses input signals via pointers, as it is now described.

Accessing Signals Using Pointers

During the simulation loop, the S-Function can access the input signals by invoking the following method:

```
InputRealPtrs uPtrs==ssGetInputPortRealSignalPtrs(S, portIndex)
```

This is done for each input port whose index is PortIndex starting at 0 (first input port). Thus, there is an array of pointers, one for each input port. To access an element of this signal, you must use the method ssGetInputPortRealSignalPtrs.

If the incoming signal at port i is complex, then the real and imaginary parts are accessed as follow:

```
real_T real_part = uPtrs[i][0]; // real part
real_T im_part = uPtrs[i][1]; // imaginary part
```

The S-Function can write to the output port by invoking the method **ssGetOutputPortSignal**. The output signal is retrieved by using the code:

```
real_T *y = ssGetOutputPortSignal(S, outputPortIndex)
```

After obtaining a pointer to a signal appearing at an output port, you can specify its real part by using the code:

```
*y++ = real_part_of_signal; // write the real part
```

Similarly, the imaginary part is specified as:

```
*y++ = imag_part_of_signal; // write the imaginary part
```

The C-MEX S-Function created for SimSyC allows a Simulink model to read and write data to MATLAB workspace while maintaining synchronization with the SystemC simulator.

3.4.3 Simulators Synchronization

The representation of simulation time differs significantly between SystemC and Simulink. SystemC is a cycle-based simulator and simulation occurs at multiples of the SystemC resolution limit. The default time resolution is 1 picosecond; this can be changed with the function *sc_set_time_resolution*. Simulink maintains simulation time as a double-precision value scaled to seconds. This time representation accommodates continuous and discrete models. SimSyC uses a one-to-one correspondence between simulation time in Simulink and SystemC. The Simulink solver is set to discrete fixed-step type, so one time step in Simulink correspond to one tick in SystemC. As mentioned previously, SystemC is the master of the simulation. Simulink is controlled from SystemC through the SimSyC interface. Using MATLAB commands *set_param* and *get_param* (with the appropriate arguments) it is possible to

```
1 // Check if simulation is running
2 engEvalString(ep, "get_param('fir16b','SimulationStatus')");
3 mxB = engGetVariable(ep, "ans");
4 string = mxArrayToString(mxB);
5 cout << "\nMatlab: simulation is " << string << endl;
```

Figure 3–9: SimSyC synchronization commands

have complete external control over Simulink. SystemC uses *set_param* to start, stop, and continue Simulink execution. Simulation is suspended at each time step by the C-MEX S-Function. For that purpose, the same command (*set_param*) is used at the end of the S-Function, but with the *pause* argument. On the other hand, SystemC requests Simulink status with the command *get_param* to synchronize both simulators. Figure 3–9 provides the SystemC code snippet of the command *get_param*.

The *set_param* command can also be used by SystemC to adjust Simulink simulation parameters between each simulation runs.

3.5 Verification Platform

SimSyC is one of the key components of the verification platform. It allows system validation and verification across abstraction levels. The primary strength of MATLAB and Simulink is not hardware verification. As mentioned, this tool is intended for algorithm development, numerical computing and data visualization. The verification platform proposed takes this into consideration and uses Mathworks' tool to assist the SystemC testbench upon request. This way the tools and languages are used for their intended purpose. Figure 3–10 gives an overview of how SimSyC is integrated in the verification flow. Since MATLAB and Simulink are used early in the design flow, it makes sense to reuse as much as possible some components

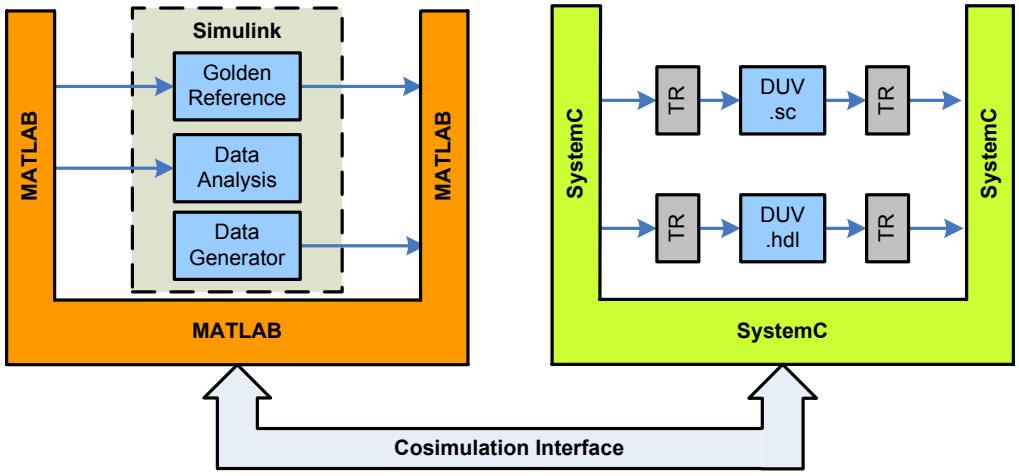

Figure 3–10: Cosimulation Interface

of these high level models to improve lower level hardware verification. The main objective with SimSyC is to simulate at lower abstraction level only those portions of the system that are part of the design. This results in faster simulation execution because the rest of the system can run at a higher abstraction level. The second objective is to provide additional flexibility and robustness to the SystemC testbench with pre-validated data generator and data analysis modules. Real life data can be quickly generated with the Simulink models using the various Blocksets available in the Simulink environment. Moreover, output data from the DUV can be forwarded by the testbench to Simulink. The verification engineer can now use graphical tools, like scopes, X-Y graphs or other mathematical operations of Simulink to further analyze the response of the system. One last benefit of using Matlab and Simulink in the verification flow is for a golden reference. A Simulink golden model can be used as a reference model by the verification system to compare the expected to the actual behaviors.

3.6 Summary

The design flow for complex numerical designs usually begins at a very high level of abstraction with tools like MATLAB and Simulink. The design flow involves the refinement of the model across multiple abstraction domains. Individual testbenches have to be created at each abstraction levels, resulting in a time-consuming and error prone process. We proposed a novel cosimulation interface called SimSyC to bridge the gap between the algorithmic domain and lower abstraction levels. This interface is used mainly in four application scenarios, connecting MATLAB and Simulink with SystemC. This constitute the basis for our verification platform.

Chapter 4
Vertical Testbench Reuse

The key element of an efficient multi-abstraction level verification environment is its reuse potential.The SimSyC interface described in chapter 3 gives the possibility to reuse Simulink stimulus and analysis modules for lower abstraction level verification. However, the SystemC testbench is specific to the design under verification and must be adapted at each abstraction level to reflect changes in the design. The purpose of this chapter is to add another element in our verification strategy to maximize the reuse aspect of the verification environment. Transactors will be inserted to enable complete vertical testbench reuse. Even though transactors are not a new strategy for verification, we found that by organizing the transactor into independent functions and by creating standard interfaces between these functions, we were able to reuse almost all the testbench across abstraction levels. Section 4.1 discusses how a layered verification environment can address complex verification projects. Section 4.2 presents the transaction-based verification technique followed with a detailed description in section 4.2.2 of how to implement transactors using SystemC. Section 4.3 presents our novel segmented adapter concept and its SystemC implementation.

4.1 Layered Verification Environment

A key objective in simulation-based verification is to find as many errors as possible within a limited time frame. Direct verification of large digital systems is almost impossible for several reasons [32]: verification tools, memory limitation, long

simulation runs, and limited controllability and observability. Consequently, large digital systems and the associated testbenches have to be decomposed into smaller components. A layered verification environment is partitioned into a set of functions that allow the overall complexity to be broken into manageable parts [9]. As with any complex design project, there are many advantages to using a structured approach for verification. Among those advantages are:

- The capacity to manage complex test scenarios.
- The reuse of verification tests and modules.
- The ability to scale projects.
- The ease of test writing.
- The understanding of functionality.
- The isolation of bugs.

This verification strategy uses a technique well known in microelectronic design: divide to conquer. By decomposing the verification environment and the design under verification into different layers, it is possible to create manageable testbenches and it is easier to detect individual bugs in smaller sub-designs before they are combined with bugs in the whole system.

A layered verification environment consists of both a layered testbench and a layered design. Figure 4–1 shows the typical decomposition used for this verification strategy. The design under verification is divided into three levels: system level, unit level, and module level. Each level has its own specifications. They can be verified individually or has a whole to insure that they meet their specifications. Hardware designers are already familiar with this kind of hierarchical decomposition.

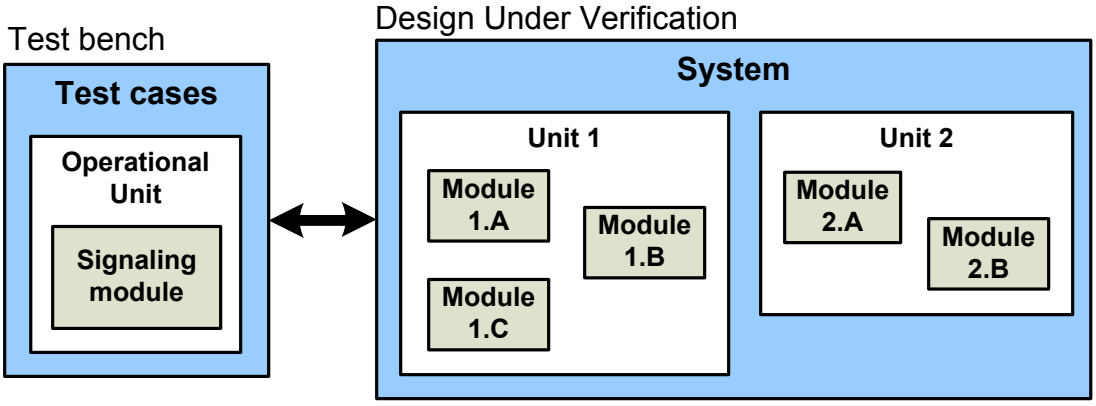

Figure 4–1: Layered verification environment

Large systems are already developed using this method. On the other hand, the testbench is usually considered as a whole. A simple testbench used to verify a small design does not requires modularity to be manageable. However, with the increasing complexity of testbenches, layered decomposition is becoming a necessity for larger design verification. The testbench in Figure 4–1 has three different levels: test cases, operational unit, and signaling unit. A new verification technique has emerged over the past few years to enable layered testbench creation. The following section introduces this technique known as transaction-based verification.

4.2 Transaction-Based Verification

The idea of transaction-based verification (TBV) is to create a verification environment that is proportionate with the system architecture under design. As the architecture of ASIC and FPGA gets more complex, the verification environment must be constructed with more abstract and layered methods to implement the verification scheme. Transaction-based verification was introduced first in 1998 [24, 20, 63]

and then included in commercial tools one year later [57, 25]. The main particularity of TBV is to begin the verification at the transaction level where the design is represented in terms of its functionality rather than specific implementation details. At this level, the design can be verified in a most effective way using transactions. The creation of test scenarios is simplified and other task like debugging and the measurement of functional coverage is facilitated. Transaction-based verification allows the verification engineer to use transactions at each stage of the verification cycle [2]. This technique is quite different from yesterday's design verification where HDL-based testbenches and waveforms were adequate to capture and debug designs. At this time, digital circuits were relatively simple hardware functions and it was possible to thoroughly verify the design with simple stimulus generators and response checkers. Today's system designs include complex hardware components and rigorous verification must be performed on a variety of modules and interfaces. System conformance of hardware-to-software (HW-SW) interfaces, signal processing and multimedia algorithms, and communications and bus interface protocols are just a few examples that illustrate the wide range of functionality that requires thorough verification. Previous verification methods are not adequate for the complexity of the design. Transaction abstraction is a promising way to capture tests and debug designs. New methods and technologies are required to construct test generators, transactors, monitors and protocol verifiers at the transaction level.

An effective transaction-level verification environment comprises several basic methods and principles [9]:

- **Separate test program content and signal-level interfaces**

  High-level sequences of instructions are used to express tests. A transactor handles the timing details associated with stimulating interface signals. More comprehensive and rigorous test suites are created by elevating test specifications to a high-level of abstraction. The time to develop and revise tests is also dramatically reduced. Transaction-level tests rely on being able to use a high-level data model to represent transaction type classes and rely on user-defined data structures. This is one reason why C/C++ has become popular for test development.

- **Constrained random test case generation**

  Design architectures are made of complex operational behaviors that have many variable parameters and operational modes. Generating comprehensive test cases is not practical and requires too much simulation time. On the other hand, generating only directed (explicit) test cases which the designer can think up and implement is too restrictive and incomplete. Using constrained random test case generation, it is possible to obtain sufficient tests without the overhead of manually generating them. The key to random verification is to define a general transaction data model that takes into account all of the possible test parameters before writing the test generator.

- **Formal protocol verification**

  Hardware processing modules are coordinated by protocols to ensure the desired outcome. Protocol verification can be a time-consuming task at the signal

level [15]. Only simple relationships between signals can be verified leaving system level relationships unchecked [9]. TBV enables protocol verification at the transaction level through passive verifiers like bus monitors and watchers. System and signal interface protocol verification is implemented using temporal function extensions to today's simulation-based verification approach. Design violations are detected continuously without requiring any additional test case development. They run in background of every simulation run, providing a comprehensive coverage.

- **Dynamic self-checking tests and transaction recording**

  The verification of complex digital systems usually involve test programs that employ tens of thousands of transactions. It is impossible to expect a human to inspect the results manually and find all possible design violations. Automated result checkers are required in order to verify correct system operation. There are two types of result checkers: static and dynamic. The former rely on current results compared against expected results generated in advance. This technique is similar to comparing with golden test vectors. Dynamic checkers use transaction recording to verify transaction results during simulation. Verification takes place for each transaction or after a pre-determined sequence. Transaction recording is used to buffer intermediate transaction results which are then evaluated by the result checker. High level verification languages provides specific functions for transaction logging and high-level data models to simplify data collection and access.

- **System level verification using HW-SW co-simulation**

  Modern complex digital systems always involve hardware and software interactions through system protocols. These HW-SW system operations often need to be modeled in a transaction-based verification environment.

In summary, transaction-based verification is a promising verification technique that increases productivity and results in more rigorous verification. Reuse is also closely integrated with this verification approach. The following illustrates a comprehensive transaction-level verification environment.

4.2.1 Transactor Structure

Transaction based verification raises the level of abstraction from signals to transactions, thus easing the development of reusable testbenches. Figure 4–2 shows a typical transaction based verification architecture. The testbench is separated into two modules: the test program and the transactor. The test program is written at a higher level of abstraction than the DUV and the transactor is the mechanism that translates the test from transactions to signals activity. A transaction is defined as a high-level data transfer characterized by its begin time, end time, and all the pertinent data related to the transaction [9]. These data represent the parameters of the transaction. As an example, a *Read* transaction will include the memory address as the transaction's parameter. This example represents a relatively simple transaction. However, a more complex transaction format will be used to describe a complete communication channel structure.

There are two key interfaces to the transactor: an interface to the design under verification and an interface to the test program. The former interface is closely

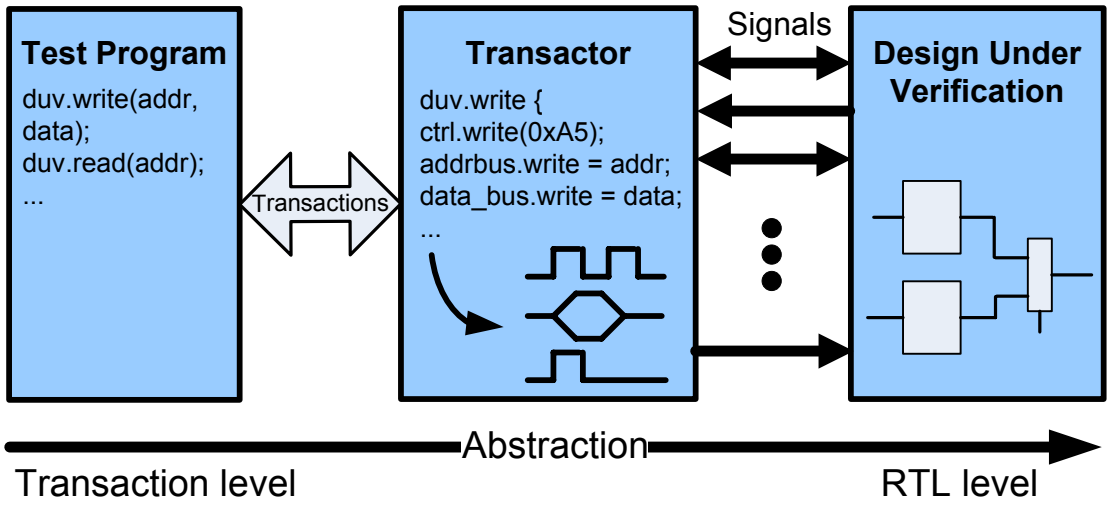

Figure 4–2: Transactor based verification

dependent on the design while the interface to the test program can be relatively generic. In addition to encoding and decoding transactions, the transactor performs the following functions:

- Negotiate the handshaking signals by which data flow is managed
- Implement clock control of the DUV

4.2.2 SystemC Implementation

A transactor can be implemented using the SystemC Verification Standard (SCV) [44]. Transaction-based verification is directly supported by SCV through a comprehensive API for transactor modeling. The following demonstrates how to create a simple transactor using the SCV modeling style [44]. Figure 4–3 shows the general architecture of a SystemC transaction-based testbench. Because SystemC is built on top of the C++ language, it provides the right mechanism to separate the testbench in a modular fashion. Using the object oriented nature of C++, the test

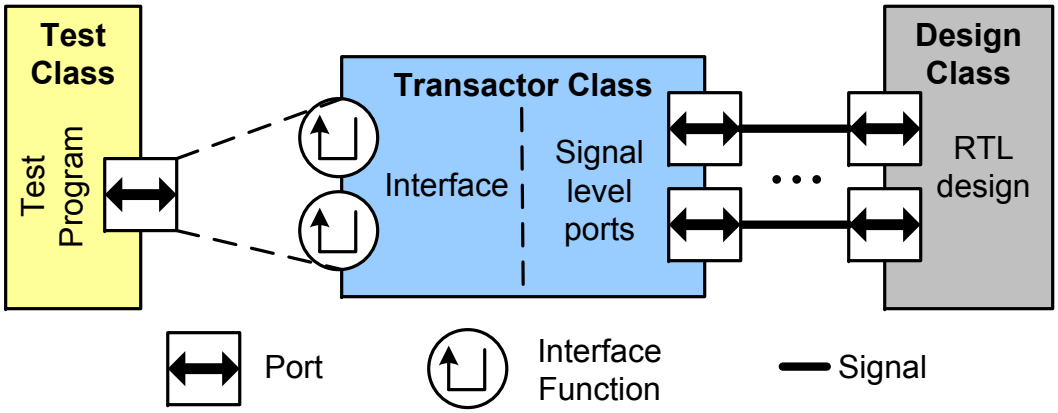

Figure 4–3: SystemC Transactor Implementation

program is encapsulated in the *Test Class*, the transactor in the *Transactor Class*, and the RTL design in the *Design Class*. In this example, the RTL design is modeled using the same language as the transactor and the test program. We will see in the next section how to interconnect a VHDL design to a SystemC transactor.

A SystemC program representing a transaction level testbench has four main components:

- The transactor's interface.
- The transactor's signal-level ports.
- The test program.
- The RTL design.

Transactor's interface

The transactor is made of two sub-classes. We can see on the left side of the transactor of Figure 4–3 the interface of the transactor. This interface will provide subroutines to be used by the test program. The transactor interface is modeled in SystemC

using C++ abstract methods. SystemC provides the class *sc_interface* from which our custom interface will be derived. The following code example shows how we create the transactor interface in SystemC:

```
class transactor_if:virtual public sc_interface{
public:
        typedef sc_uint<16> address_type; // ‘‘Address’’ data type
        typedef sc_uint<8> data_type;     // ‘‘Data’’ data type

        struct write_type{              // Write operation format
               address_type addr;
               data_type data;
        };
// Read method
        virtual data_type read(const address_type*) = 0;
// Write method
        virtual void write(const write_type*) = 0;
};
```

In this code snippet, two virtual methods are declared for this transactor: a *Read()* and *Write()* methods. These subroutines will be available to the test program and will abstract away all lower level details.

The transactor's signal-level ports

On the other end of the transactor (right side) depicted in Figure 4–3 are the signal-level ports. These ports handle the communication between the RTL design and the transactor. They are captured in another class named *design_ports*:

```
class design_ports:public sc_module{
public:
        sc_in<bool> clock;           // Design ports definition
        sc_inout<bool> rw_ctrl;
        sc_inout<bool> addr_req;
        sc_inout<bool> addr_ack;
        sc_inout<sc_uint<8>> addr_bus;
        sc_inout<bool> data_ready;
        sc_inout<sc_uint<8>> data_bus;
 };
```

This class is closely dependent on the design. Any changes on the design interface will have to be reflected in this class. All the signals in this class correspond to the interface of the DUV used to communicate with the external environment.

The next step is to create the transactor with these two sub-classes. A new class is defined using inheritance; a key feature of the C++ language. The *transactor* class is derived from the *transactor_if* and the *design_ports* classes:

```
class transactor:public transactor_if,public design_ports{
public:
       SC_CTOR(transactor){}
```

```
        virtual data_type read(const address_type *);
        virtual void write(const write_type *);
};
```

We now have a transactor definition with an interface to communicate at the transaction level with the test program and a collection of signal-level ports to communicate at the RTL level with the design under verification. The content of the *read()* and *write()* functions have to be defined according to the design operational mode. These functions will convert the transactions into signal-level activities. The detailed implementation of *read()* and *write()* is not shown here in this simple example, but a more complete running case study is presented in chapter 5.

The test program

The test program can now be written at the transaction level rather than RTL level. With all the signals abstracted by the transactor, the verification engineer can write a more efficient test program focusing on important system behaviors. The next code snippet shows a simple test program using a counter.

```
// Testbench module definition
class testbench:public sc_module{
public:
        // The testbench has only one port
        sc_port<transactor_if>transactor;

        // to communicate with the transactor
        SC_CTOR(testbench){SC_THREAD(main);}
```

```
       void main();
};

// Testbench main() function definition
void test:main(){
     for(char i=0; i<10; i++){  // Generate 10 addresses
         transactor_if::address_type address = i;
         // Next address to read from

         // Read data from the address' location
         transactor_if::data_type data = transactor->read(&address);
         // Print the result to the screen
         cout << ''Read result:'' << data << endl;
         }
}
```

This test program generates ten read transactions and sends them to the transactor one at a time. The transactor will convert each read request by stimulating the proper signals of the design under verification. The data is then collected by the transactor and sent back to the test program.

The RTL design

The design under verification is a register transfer level (RTL) model of the circuit. It has the same signal-level ports used by the transactor. Therefore, the *duv* class is

derived from the *design_ports* class previously created for the transactor. Here is an example of the code for the RTL design class:

```
class duv:public design_ports{ // Class duv derived from design_ports
              ...              // Other variables go here
public:
       SC_CTOR{duv};
           SC_THREAD(addr_cycle);  // Address cycle thread
           SC_THREAD(data_cycle);  // Data cycle thread
           ...                    // Other functions go here
       }
       //Address cycle operations go here
       void addr_cycle(){while(1) ...}
       //Data cycle operations go here
       void data_cycle(){while(1) ...}
};
```

This simple example has demonstrated how SystemC offers the right mechanisms to create efficient transactors. More complex transactors can be built using the same technique presented above.

4.2.3 Limitations of transactors

Transaction-based verification aims to facilitate functional verification of RTL designs by raising the testbench to a higher level of abstraction. A transactor is required to bridge the abstraction gap between the testbench and the design. Transactor is an efficient solution for the verification of complex RTL designs. However,

transactors have some intrinsic limitations that make them difficult to use in a multi-abstraction level verification platform. We have identified three major limitations to transactors:

- Applicable only to RTL designs.
- Impossible to reuse.
- Design language specific.

A transactor makes the connection between an RTL design and the testbench. As a first limitation, this implies that the verification begins late in the design flow, when an RTL version of the design is available. However, as presented in section 3.1 the design of complex digital system begins at a higher level of abstraction and is refined down to the RTL level. Thus, to be efficient verification must also start early in the design flow and follow the design across abstraction levels. Transactors do not provide the right mechanism to handle this kind of vertical verification.

The second limitation of transactors is related to their reuse capability. Since the transactor has a signal-level interface to communicate with the design, it is impossible to reuse it, even if the other design has similar functionalities but a different signal-level interface. For example, two equivalent IP core designs, one using an AMBA interface [19] and the other a Core Connect [28] interface, will each have their own transactor; even if the design and the testbench are the same. The transactor behavior is equivalent, but we cannot reuse it without major modifications to its internal structure.

Finally, transactors are specifically created to communicate with the language used for the design. As an example, the same design represented with two different

languages will have two dedicated transactors. Therefore, a more generic interface is required in a multi-abstraction level framework, where the design can be represented with different languages.

Transaction-based verification is an efficient verification technique. However, it cannot be integrated in our multi-abstraction level verification platform according to the limitations of transactors presented above. In the following section, we proposed to generalize the transactor concept to overcome its' limitations and make it integrable in our verification framework.

4.3 Transactor Migration Problem

A designer must be able to migrate transactors with minimal modifications to be able to connect it to different simulation tools and abstraction levels. When migrating a transactor, some changes are required to re-host the transactor. This is different from reuse where the whole transactor can be reused without modifications. To recall section 4.2.3, the actual organization of a transactor does not allow easy migration. Several limitations confine transactors to a single simulation tool and abstraction level. One way to migrate a transactor, without the need to completely re-write a new one, is to generalize its definition to create a more flexible solution. This new adapter will be integrated in our multi-abstraction level verification platform.

4.3.1 Requirements

The design flow presented in section 3.1 shows that the design is being refined across abstraction levels using different tools and languages. Therefore, an adapter is required between the testbench and the design to make them compatible. This adapter has to be flexible so it can be configured throughout the refinement process.

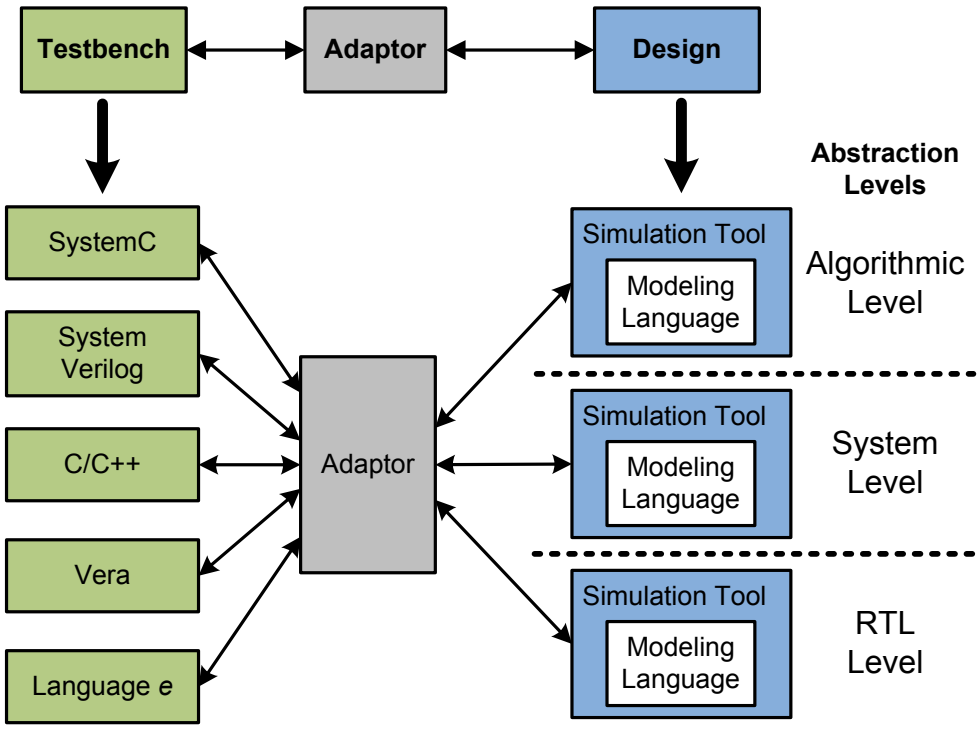

Figure 4–4: Adapter Configurations

The diagram of Figure 4–4 shows the different configurations taken by the adapter. On the left side, the adapter communicates with the testbench, which can be written using different high level verification languages. On the right side, the adapter communicates with the design under verification, which can be modeled using different languages and run on different simulation tools. At each abstraction level, several modeling languages and tools can be used to represent the design.

The goal of the adapter is to provide a configurable bridge between the testbench and the design under verification. As the design is being refined across abstraction levels, the adapter migrates in the same direction to provide the right communication

mechanism. Therefore, the testbench can be created early in the design process and reused at each abstraction level.

The adapter to be designed will have two main functionalities. It will:

1. Link the testbench to the design simulation engine.
2. Translate data across abstraction levels.

Depending on the simulation scenario, these functionalities may be implemented or not. If both functionalities are not implemented, the adapter is not required at all. For example, a VHDL testbench used to verify a VHDL design does not require this kind of adapter. The same simulation engine is used and both the testbench and the design are represented at the same abstraction level. If we raise the testbench to the transaction level using the SystemC language, the adapter will have to first, link together both the SystemC and the VHDL simulation engines, and then translate SystemC transactions into VHDL signals. In this example, the adapter is thus a transactor, as defined in the previous section.

With these configurations defined, the internal components of the adapter begin to emerge. As shown in Figure 4–5, the adapter will have two distinct link layers and a central translator module. It is important to recall at this point that each of these components may be optional depending on the simulation scenario.

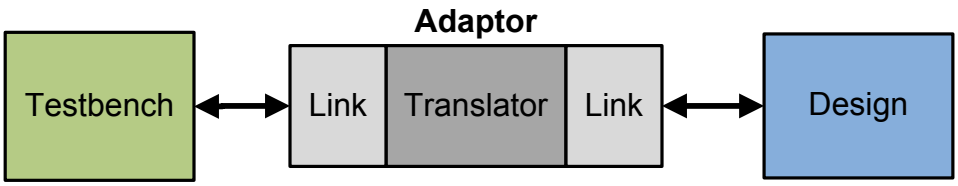

Figure 4–5: Adapter components

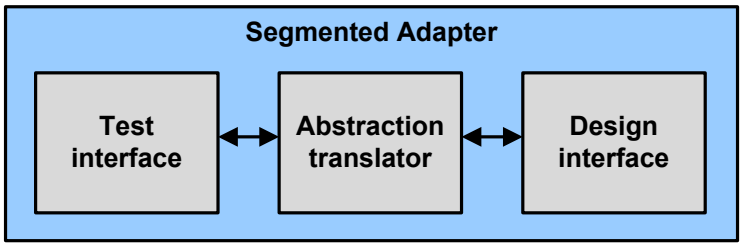

Figure 4–6: Segmented adapter

4.3.2 Segmented Adapter

Considering the requirements in a multi-abstraction level verification framework we proposed a novel solution, called segmented adapter, to link the design under verification to the testbench across abstraction levels. Based on the requirements presented in section 4.3.1, a segmented adapter is divided into three segments, as shown in Figure 4–6. The test interface segment (TIS), the abstraction translator segment (ATS), and the design interface segment (DIS). Each segment has a specific role according to the following definitions:

- **Test Interface Segment**

  The TIS has an external port to communicate with the testbench. The main function of this segment is to provide the proper communication link and a set of high level routines accessible by the testbench.

- **Abstraction Translator Segment**

  The ATS has only two internal ports to communicate with the TIS and the DIS. This segment role is to translate high level commands received from the TAS into a lower level representation that will match with the design. The ATS achieves this by manipulating the design interface segment.

- **Design Interface Segment**

  The DIS has an external port to communicate with the design under verification. This segment is closely related to the design. It implements the communication details to connect the adapter with the simulation medium through abstraction-specific objects.

A segmented adapter has several advantages over traditional transactors. Firstly, its modular organization enables the substitution of individual segments. As the design is being refined across abstraction levels, only the design interface segment needs to be replaced to reflect the abstraction changes. Figure 4–7 shows how a segmented adapter can be migrated easily. In this example, the same segmented adapter is being used to verify two representations of the same design under verification. To migrate the transactor between the SystemC and the VHDL description of the DUV, only the design interface segment needs to be changed while the ATS and TIS remain the same. In some cases, the abstraction translator segment might also be replaced if the abstraction level of the DUV has changed significantly.

4.3.3 Platform Integration

The segmented adapter construct has been integrated in our verification platform to link together the design and the testbench across abstraction levels. Figure 4–8 shows how we used SystemC to implement individual segments of the adapter. It relies on SystemC interface specification. All segments are connected together using abstract interfaces and ports. The modular organization favors the replacement of single segment.

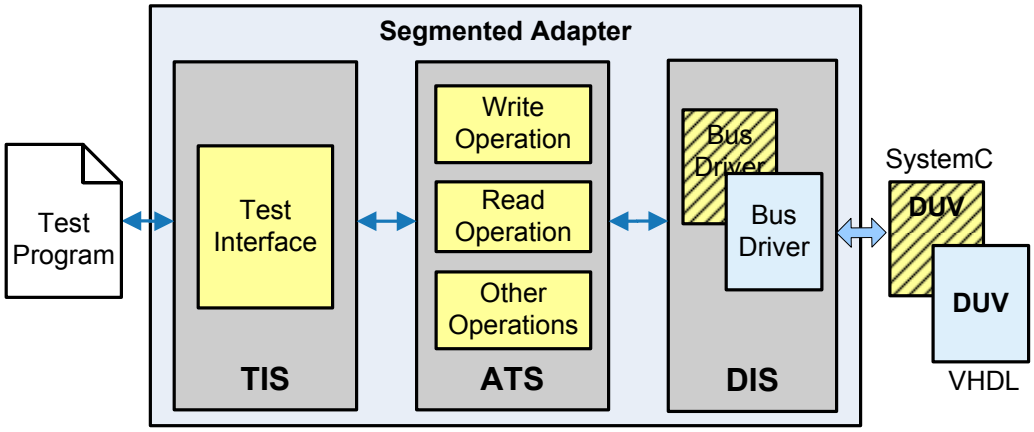

Figure 4–7: Segmented adapter example

4.4 Summary

The validation and verification of complex systems involves the creation of multiple testbench components. Reuse is an important issue when it comes to verification time and efforts. We proposed an evolved version of traditional transactors called segmented adapters. This mechanism is inserted between the testbench and the design and enables vertical reuse across abstraction levels. The segmented adapter is partitioned into three segments that can be changed independently according to the abstraction level requirements. SystemC provides the right structure to efficiently implement segmented transactors with the interface specification.

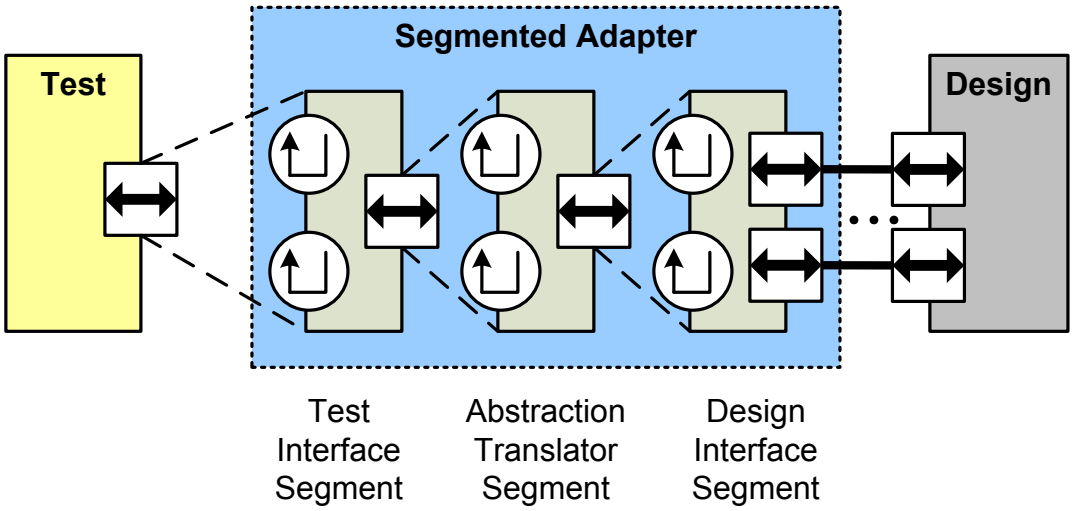

Figure 4–8: Segmented adapter implementation

Chapter 5
Case Study

The verification platform presented in this dissertation has been evaluated through the verification of two telecommunication design examples. The key components of the platform have been validated independently as a proof of concept. The first example is a Manchester encoding system that will be verified with MATLAB/Simulink through the SimSyC cosimulation interface. The second example is a multi-equalizer design used in a software defined radio. It is important to consider at this point that the verification platform proposed is not restricted to this kind of circuits. It can be used for the verification of all kind of complex digital design that are represented at multiple levels of abstraction.

5.1 Manchester Encoding System

To evaluate the SimSyC cosimulation interface presented in this work, a Manchester encoding system with clock recovery capabilities has been used as a first case study. An existing Manchester example from Mathworks [37] has been adapted for our needs. Manchester encoding is a technique employed to modulate digital data transmitted over a physical communication channel [21]. This technique is widely used in Ethernet applications to convert baseband digital data into an encoded waveform with no DC component. A particularity of this technique is that the clock can be extracted from the received data, so a separate clock is not required. Figure 5–1

shows a typical Manchester encoder-decoder configuration. For this example, the objective was to design a hardware version of the decoder module.

5.1.1 Simulink modeling

As a first step, a Simulink model of the entire system has been created. This high abstraction level representation was used to validate the algorithm of the decoder. A delay lock loop (DLL) will monitor the received data to maintain synchronization. The decoder will oversample the received signal with a clock period of 1/16th the encoder clock. Original data will be extracted from this signal without any additional information about the transmitter module. The DLL will manage the differences between the clock used to encode the signal and the clock in the receiver module. By adjusting the period of the receiver clock, minor differences between the transmitter and receiver clock will be compensated. Drifts in the channel delay will also be corrected.

Figure 5–2 shows the Simulink model created. The Manchester decoder block is an M-Sfunction written using the MATLAB language. To verify this block, a Manchester encoder will generate a continuous data stream. This encoder has a *phase offset* input connected to a phase and frequency error controller to disturb the transmitted signal. Performances of the Manchester decoder will then be analysed with multiple

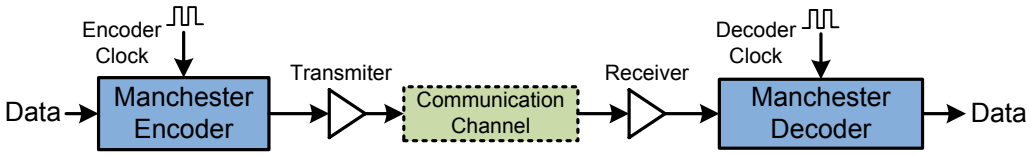

Figure 5–1: Manchester encoding system

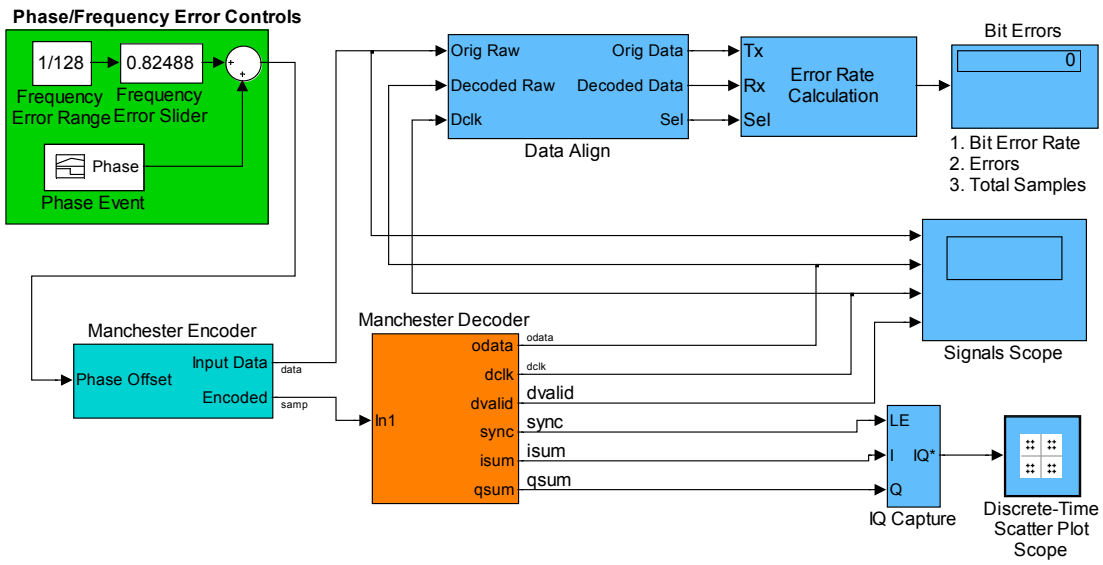

Figure 5–2: Simulink model of the Manchester encoding system

analysis modules. Bit error rate is measured by comparing the decoded data streams to the original ones. The Simulink communication blockset library is also used to display the results graphically to facilitate performance analysis. A discrete-time scatter plot scope displays useful information about the signal to reveal the modulation characteristics, such as pulse shaping or channel distortions of the signal. Finally, a signal scope displays binary signals in a timing diagram format. Several simulation runs were performed to verify each component of this high abstraction level model and to validate and optimize the Manchester decoder algorithm.

5.1.2 SystemC decoder modeling

Once the high level model has been extensively verified, we can proceed to a lower abstraction level implementation. For this example we choose the SystemC language to implement the decoder module. With SystemC it is possible to refine

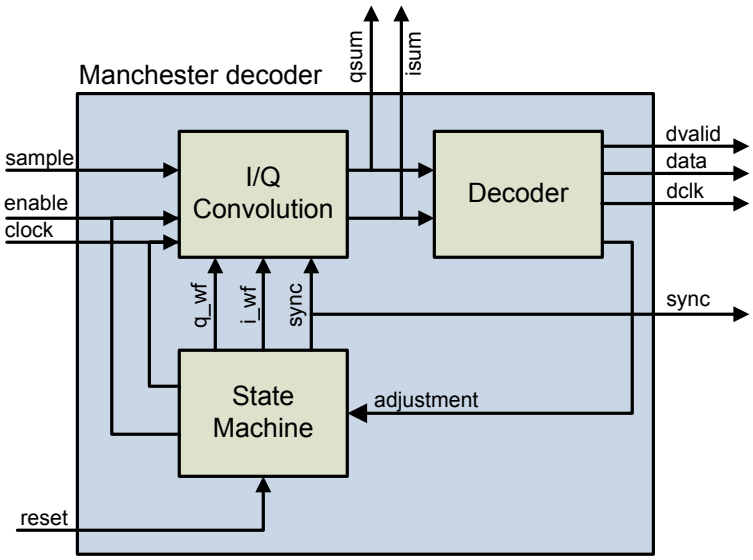

Figure 5–3: SystemC Manchester decoder

the model down to the register transfer level and then synthesize the design into real hardware. In addition, the SimSyC interface will be used to verify the design by reusing already created Simulink data generator and analysis modules. The SystemC Manchester decoder is composed of three modules: the I/Q convolution, the decoder, and the state machine. The bloc diagram of this SystemC design is depicted in Figure 5–3. The decoder extracts the data and the original clock in a two step process. The convolutions of the in phase *i_wf* and quadrature *q_wf* signals are calculated, and the results are processed by the decoder combinational circuit. The state machine generates the *i_wf* and *q_wf* signals and makes the proper correction to the phase errors with the adjustment factor provided by the decoder.

The SystemC Manchester decoder model has been created using Microsoft Visual Studio 2005. Programming a SystemC model is similar to pure C++ programming. Visual Studio provides all the tools required to edit, compile, link, and debug the program. Since the program is linked with the SystemC library, the final executable file includes the OSCI simulation kernel. A simulation run is carried out simply by running the executable file. As can been seem in Figure 5–4 the output window of a SystemC simulation run is quite simple. Only basic program debugging can be done. More advanced digital system verification is possible with our novel cosimulation interface SimSyC.

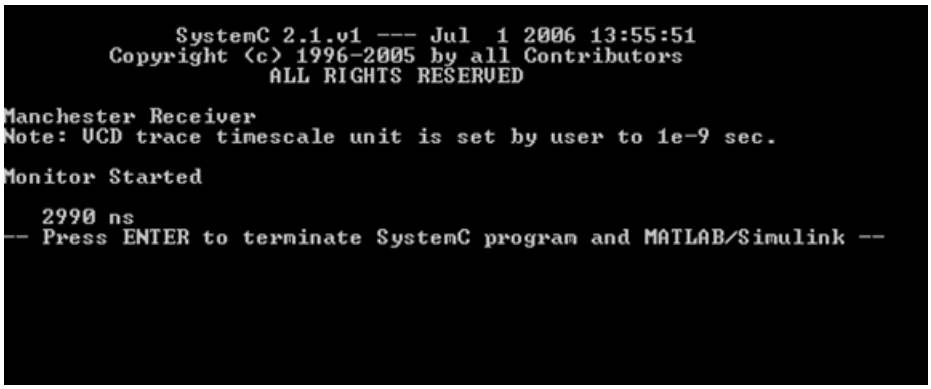

Figure 5–4: SystemC output window

5.1.3 SystemC decoder verification using SimSyC

The verification of the Manchester decoder SystemC model has been done with SimSyC. To verify the correctness of our design, a continuous flow of encoded data is required. Perturbations must be applied to this data stream to assess the performances of the decoder. Analysis requires the proper monitoring modules to throughly measure and evaluate the decoder operational behaviors. It is possible to manually

program the data generator and analysis modules using the SystemC language. However, it will necessitate a lot of time and the process is error prone. SimSyC is an efficient solution to this problem. Figure 5–5 shows how SimSyC is used in this example.

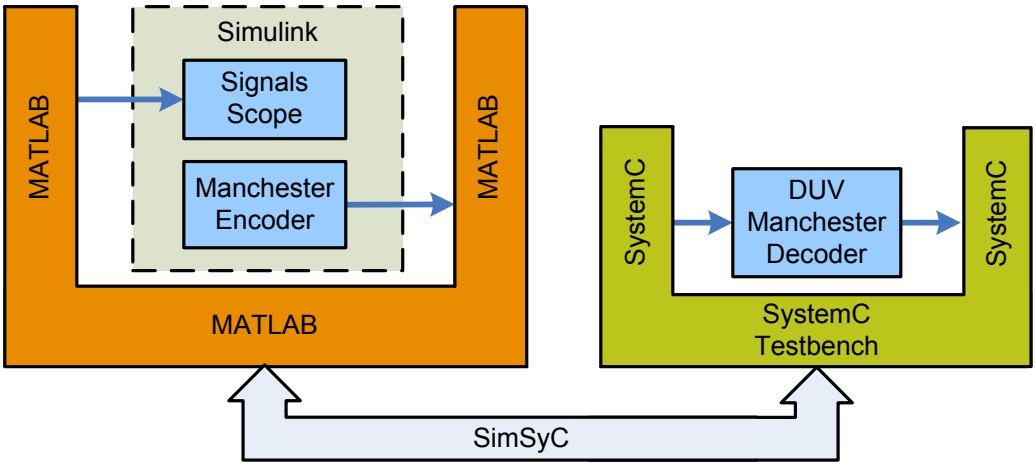

Figure 5–5: Manchester example verification framework

The design under verification (DUV) is surrounded by a SystemC testbench. SystemC is the central component of the verification platform. All the verification scenarios are written in SystemC. For simple designs, the SystemC testbench can be sufficient for a complete verification. In this example, we used the SimSyC cosimulation interface to extend the testbench capabilities. With SimSyC, MATLAB and Simulink can be used to assist the SystemC testbench. Since we already have a Simulink model of the entire Manchester system, we can quickly reuse this model as a data generator and data analysis unit. Figure 5–5 shows how the data path between the Simulink model and the DUV. In Simulink we have the Manchester encoder that generates the data stream for the SystemC testbench. Outputs from the

DUV are analyzed with the existing Simulink visualization blocks previously created. SimSyC serves as a data transport layer to exchange data back and forth between Simulink and SystemC. The complete simulation execution flow is shown in Figure 5–6.

The SystemC testbench is composed of severals modules as shown in Figure 5–7. The controller module is in charge of the simulation execution. Initialization, synchronization, and verification commands are issued by this module. The monitor unit acquires a series of signals from the DUV and then regroups them for further analysis by Simulink. The SimSyC module encapsulates the SystemC part of the SimSyC interface. It is connected to the testbench and the monitor so it can generate data when requested by the testbench and proceed to the analysis of the data coming from the monitor.

As can been seen in Figure 5–8 the Simulink model is almost the same one that was used previously. Only the Manchester decoder block has been replaced by the Simulink part of the SimSyC cosimulation interface. To recall from Chapter 3, this block is a MEX S-Function written using the C language that can be customized according to the inputs and outputs of the current design. In this example, all the signals captured by the SystemC testbench are mapped to the Simulink SimSyC block so they can be routed to the different analysis elements.

The verification of the SystemC Manchester decoder was done by running multiple simulation runs. For each simulation, different testbench scenarios were tried. All the test scenarios are written using SystemC. The Simulink model never has to

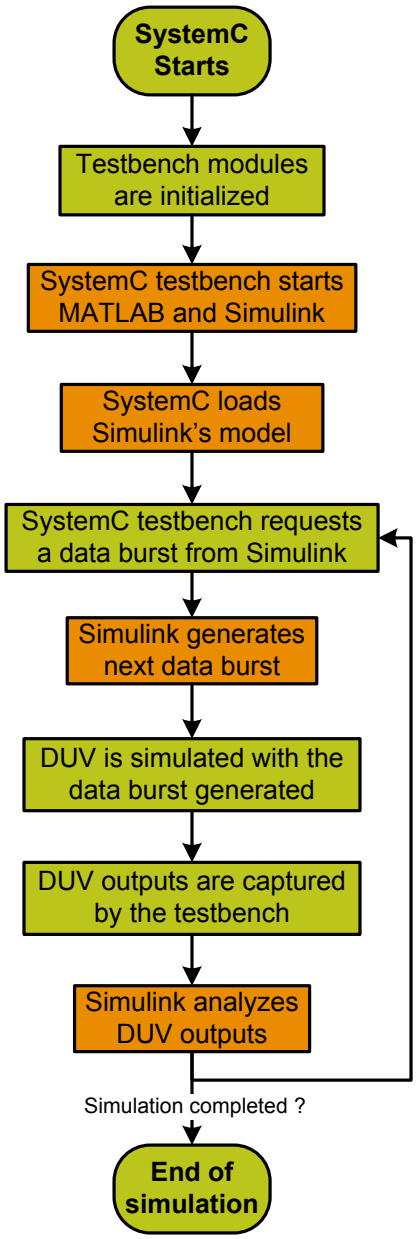

Figure 5–6: Simulation execution flow

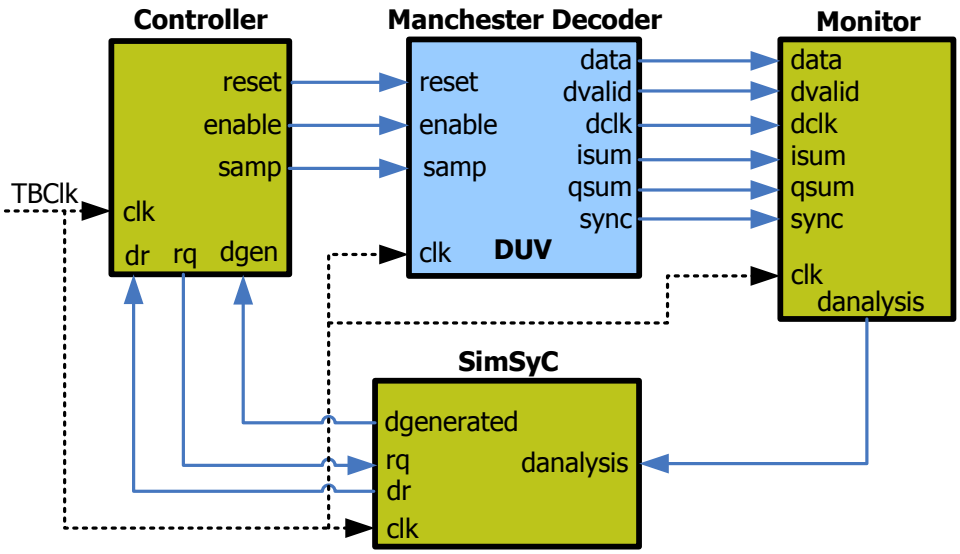

Figure 5–7: SystemC testbench

be modified manually since all Simulink parameters, including individual block variables, can be changed from the SystemC testbench. For example, the phase/frequency error controls in Simulink are initialized by SystemC, through the SimSyC interface, before each simulation run. This simplifies the testbench scripting, since all the simulation parameters are unified in the same language and in the same place. Figure 5–9 shows an example of the graphical user interface available to the verification engineer during a simulation run. All experiments were run on a 2.4 GHz Pentium 4 computer with 1G RAM, running Windows XP. We can see in the background the Simulink model running and in the foreground we have the signal scope, the discrete-time scatter plot, and the SystemC output window. Experimental results show that the SimSyC cosimulation interface significantly reduces the time spent on testbench creation when the design has to be refined across abstraction levels. The reuse of

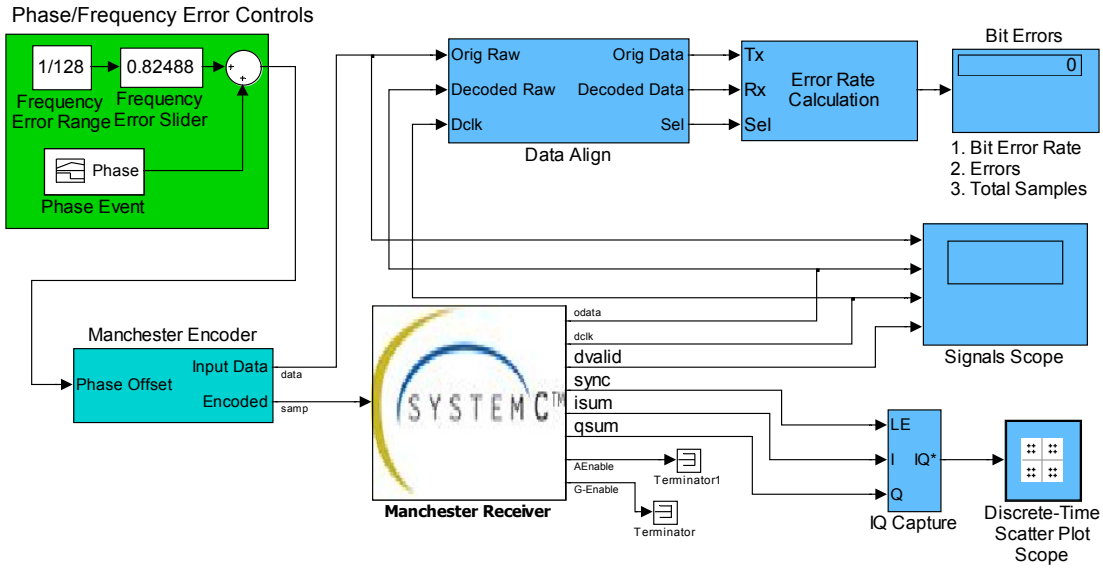

Figure 5–8: Simulink model with a SystemC manchester decoder

high-level Simulink models to assist a SystemC testbench is now possible. The time spent on verification is thus reduced and the quality of the testbench is increased. By reusing already verified components, the verification effort is less prone to errors. This example has been used as a proof of concept of the SimSyC interface. SimSyC can be scale up without problems for larger designs.

5.2 Software Defined Radio Multi-Equalizer Architecture

A second case study as been developed as a proof of concept for the segmented adapters mechanism and the SimSyC interface. The verification platform presented in this work has been used by a design team located at the Ecole de Technologie Superieure, in Montreal. They worked on a project called MAME [50] (Methodology and Architecture of a Multi-Equalizer). This team is looking for new design methodologies and architectures to build a multi-equalizer; one of the key components in the

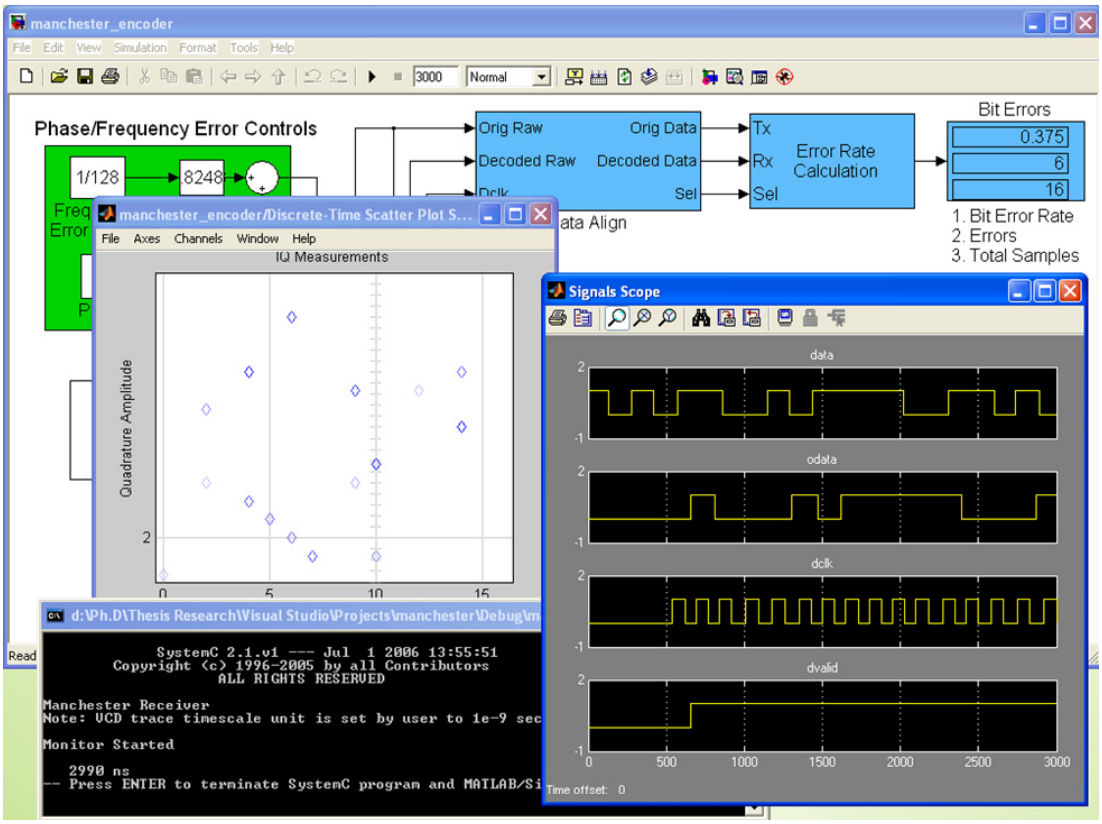

Figure 5–9: Simulation run example

receiver module of a software defined radio [41]. Our proposed verification platform and methodology have been designed to cooperate seamlessly with this design flow.

5.2.1 SDR Design Challenge

A Software-Defined Radio (SDR) is a combination of digital filters, analog components and processors, each requiring different design approaches with a different tool or language. As an ideal candidate for System-on-Chip implementation, SDR combines heterogeneous processes operating within strict functional and physical constraints. The design of such complex systems requires traditional VLSI tools to be complemented by an instruction set simulator. In addition, high-level specifications and early prototyping of the system is mandatory in order to evaluate risks and detect design flaws. Problems arise when the variety of tools and languages used to develop individual SDR modules must be interconnected in order to build a functional prototype. Up to half a dozen specialized tools and languages may be involved, depending on the heterogeneity of a design [36]. Further complications arise when the design goes through refinement and hardware/software partitioning iterations. Re-design is often necessary across abstraction levels. The need for a holistic approach is imperative. This approach must preserve tool heterogeneity while providing a design path from specifications to implementation. A design and verification methodology has been developed for an important SDR subsystem, the equalizer. This multi-language based methodology uses the Unified Modeling Language (UML) to build an executable specification with stem connections to SystemC, Simulink and simulation tools. Validation and verification is enabled at all points in

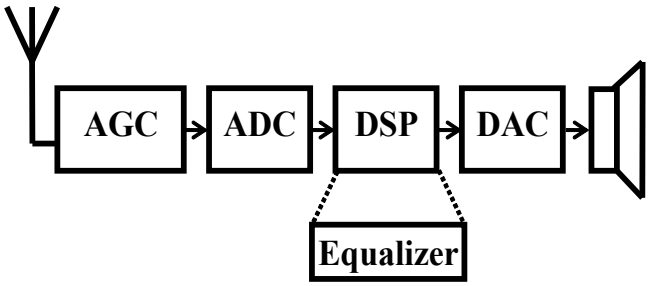

Figure 5–10: Software defined radio architecture

the design, through segmented adapters and SimSyC, in order to assert specification conformance and constraint compliance.

5.2.2 SDR Architecture

An SDR is a wireless communication device in which the physical and link layer functions are implemented or maybe configured by software [41]. Basically, a wideband ADC or DAC is placed as close as possible to the antenna, so most of the data processing is done by DSP techniques. This adds more flexibility to the radio so it can be easily reprogrammed to support different modulation, coding or access protocols. Figure 5–10 shows a simplified diagram of the architecture of an SDR receiver. The received signal is digitalized by the Analog-to-Digital Converter (ADC) just after the Automatic Gain Control (AGC) stage. Next, the Digital Signal Processing (DSP) block may be implemented with a Digital Signal Processor or with a combination of a processor and FPGAs. One of the key elements of the DSP block is the equalizer. In order to compensate the phase and amplitude distortion of the signal that occurs during transmission over the channel, an equalizer is included in the data processing flow [49]. There are three main equalizer architectures used today;

the linear equalizer [49], the decision-feedback equalizer [56] and the interference canceler [23]. Theses equalizers are made of adaptive digital filters, whose coefficients are updated according to specific algorithms [31]. These algorithms depend on the impulse response of the channel [51]. The difficulty with equalization is the strong dependency between the equalizer architecture/algorithm and the telecommunication scenario. In other words, the best equalizer architecture changes over time and with receiver location. Another concern is that an equalizer usually combines fast operations, that are performed at Intermediate or Radio Frequency, with much slower feedback loops. The coexistence of these different clock domains, which differ by orders of magnitudes in frequency values, involve a very rigorous approach to the design and verification process. Considering the different equalizers to implement, the rigor of this methodology should not entail a reduced flexibility in terms of hardware-software partitioning and migration of the algorithms.

5.2.3 MAME's Design Flow

The design flow used in the MAME project is shown in Figure 5–11. The flow is divided into three main design steps, distributed across multiple levels of abstraction. The first step is to build a Platform Independent Model (PIM) of the system. Next the model goes through a partitioning process to generate a Platform Specific Model (PSM). This PSM is then refined down to the RTL level for integration on the target board. Our verification environment is connected to this flow from the early design steps and reused throughout the flow.

After the preliminary modeling with UML, an algorithmic model of the multi-equalizer is created with Simulink. This model is simulated with another model

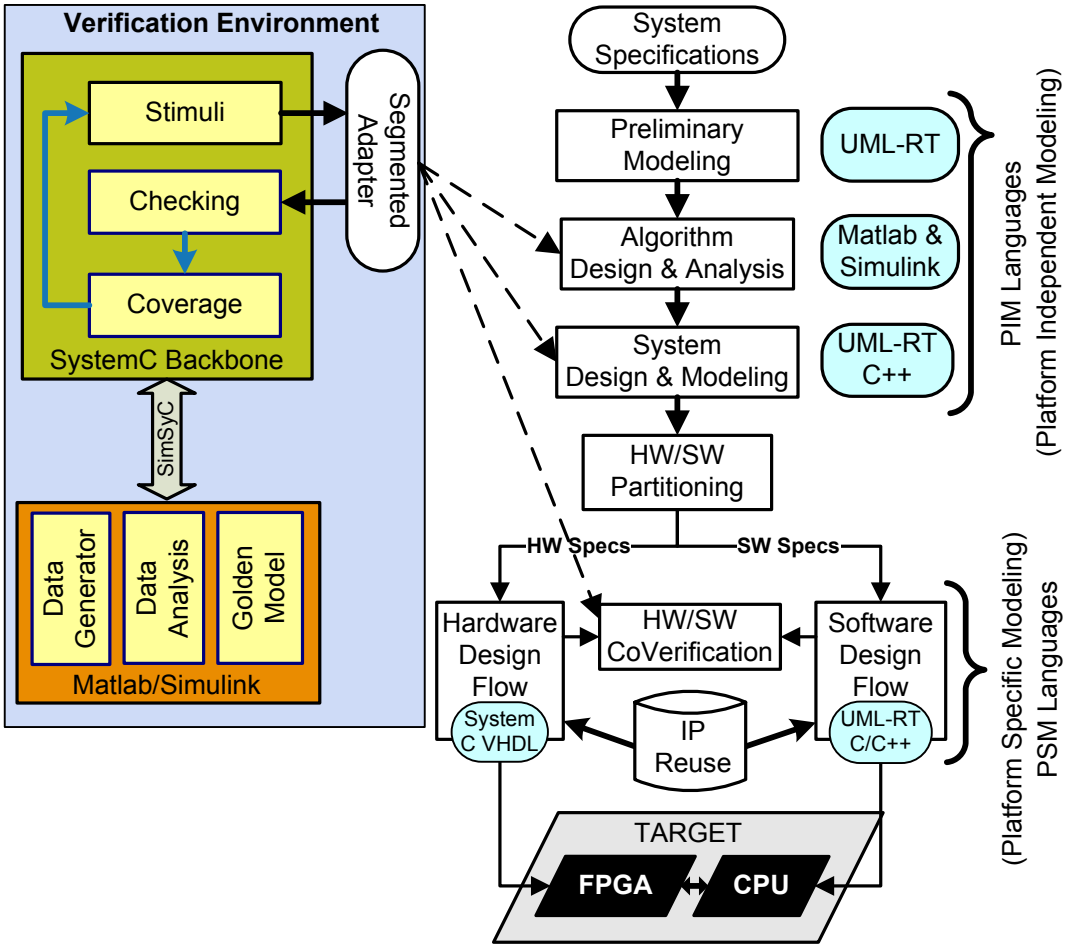

Figure 5–11: MAME's Design Flow

representing the conditions in which the multi-equalizer operates. At this early stage in the design process, the SystemC verification environment is developed. A segmented adapter is defined between MATLAB/Simulink and SystemC. A design interface segment for MATLAB (DIS.mat) is created. This *DIS.mat* uses SimSyC as the communication layer. The algorithmic model of the multi equalizer is then validated and optimized using the SystemC verification system. Multiple test scenarios

are written in SystemC and the simulation execution is controlled from the SystemC testbench. For example, the testbench will change dynamically the communication channel model to check how the multi equalizer behaves.

Once the multi equalizer algorithm satisfies the requirements of the specifications, the next step in the design process is to create a system level model of the architecture of the multi equalizer. SystemC is used for this design step. In order to reuse the same verification system, the MATLAB *DIS.mat* is replaced with a SystemC version called *DIS.sys*. This design interface do not has to use any special communication mechanism since both the design and the testbench are written using SystemC. To efficiently validate and verify the SystemC model of the multi equalizer, the verification environment reuses the Simulink model of the operating conditions through the SimSyC cosimulation interface. Realistic equalization scenarios were produced to stimulate the design. While Simulink provided the symbol train to equalize, the real-time performance of the executable model was profiled. This provided guidance regarding which parts of the equalizer would be implemented in hardware.

The last design step is the creation of a VHDL model of the selected SystemC modules that will be implemented in hardware. The same verification environment is reused by changing the *DIS.sys* with the *DIS.vhd*. The Simulink model of the operating conditions is also reused is a similar way as in the previous step. Figure 5–12 shows the general organization of the verification platform. The evolution of the platform is indicated by the encircled numbers, representing the three majors steps of the verification process, and is summarized below:

Step 1
- The multi equalizer and the operating conditions are modeled using Simulink
- A segmented adapter is created using SystemC
- The SystemC verification environment is defined
- Preliminary verification of the Simulink algorithmic model is performed by the SystemC verification system through the segmented adapter

Step 2
- The multi equalizer architecture is modeled using SystemC
- The segmented adapter is migrated to a lower abstraction level by redefining the DIS segment
- The Simulink model representing the operating conditions of the multi equalizer is reused by the SystemC verification environment throught the SimSyC interface
- Verification of the SystemC architectural model is performed by the SystemC verification system through the segmented adapter

Step 3
- The multi equalizer hardware components are modeled using VHDL
- The segmented adapter is migrated to a lower abstraction level by redefining the DIS segment
- The Simulink model representing the operating conditions of the multi equalizer is reused by the SystemC verification environment throught the SimSyC interface
- Verification of the VHDL models is performed by the SystemC verification system through the segmented adapter

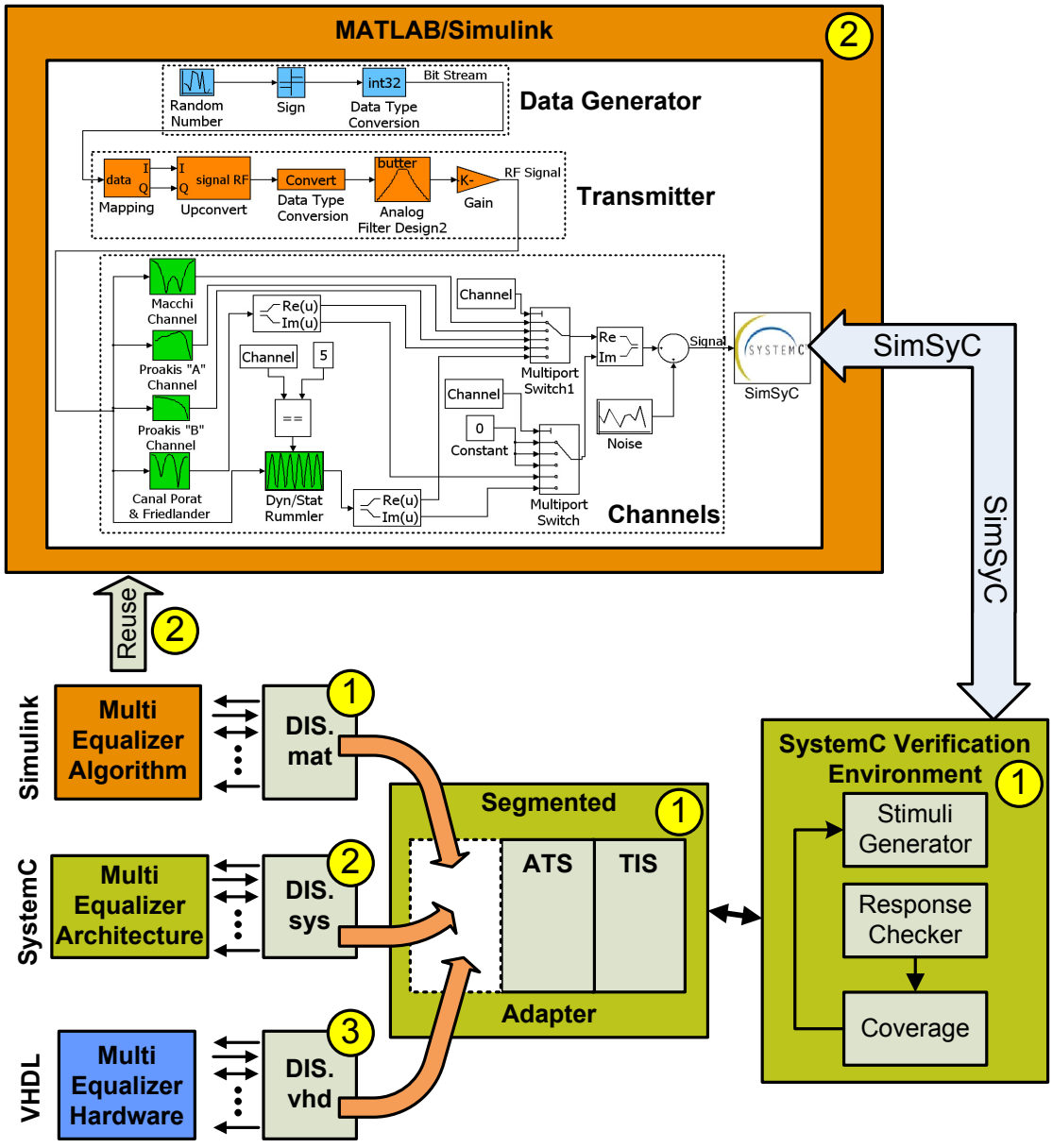

Figure 5–12: Multi-equalizer Verification Framework

Abstraction Level	Design	Verification	Total
Algorithmic	14	25	39
System	21	21	42
RTL	25	25	50
Total	60	71	131

Table 5–1: Design effort using the improved design flow (days)

5.2.4 Results

The verification platform has been used throughout the design flow to detect design flaws and specification violations as soon as possible. The advantage of this platform compared to other verification solutions, is the possibility to reuse the high level algorithmic model to assist the verification environment. The design of the multi-equalizer took approximately 131 days (6.5 months) to complete with two engineers working full time on the project. Table 5–1 summarizes the breakdown of the design time across the whole project. At the algorithmic level, it took 14 days to create the Simulink model used to design and optimize the algorithm of the multi-equalizer. Next, a testbench has been created and multiples simulation runs where conducted to verify the model. This verification process took approximately 25 days for a total of 39 days. At this level of abstraction, the verification platform does not provides any additional gain. As the design is being refined at the system level and the RTL level, the verification platform efficiency begins to pay off. As we can see from Table 5–1, the verification effort is equal to the design effort.

Using a traditional design flow, where the verification effort represents 70% of the total design time, will yield in more time spent on testbench development and simulation runs. Table 5–2 shows the expected design effort repartition using a

Abstraction Level	**Design**	**Verification**	**Total**
Algorithmic	14	25	39
System	21 (30%)	**49** (70%)	70
RTL	25 (30%)	**58** (70%)	83
Total	60	**132**	**192**

Table 5–2: Design effort using a traditional design flow (days)

traditional design flow with 70% of the design time spent on verification (at the system and RTL levels). The result is 192 days as the total developpment time for this project, compared to 131 days using the improved design flow. This represents a productivity gain of around 32% over a traditional design flow that has limited testbench components reuse and software interroperability.

Figure 5–13 shows graphically how the vertical testbench reuse strategy reduces the development time of a design. Using a traditional design flow (upper half of Figure 5–13), the verification effort of this project counts for 70% of the total design time at each abstraction level. Using the improved design flow (lower half of Figure 5–13), the verification effort gets reduced to 50%. Testbench development time is considerably reduced by reusing testbench components already created at higher abstraction level. This results in a productivity gain of around 32%. The whole verification process is also less error prone by eliminating manual intervention. The testbench is not anymore subservient to the design efforts, but it is now part of it.

The SimSyC interface enables verification strategies that were not possible before. Figure 5–14 compares the verification productivity gain of SimSyC against HVL and HDL testbenches. With SimSyC, it is possible to reuse a complete model of the environment of the design, right at the beginning of the verification tasks. The

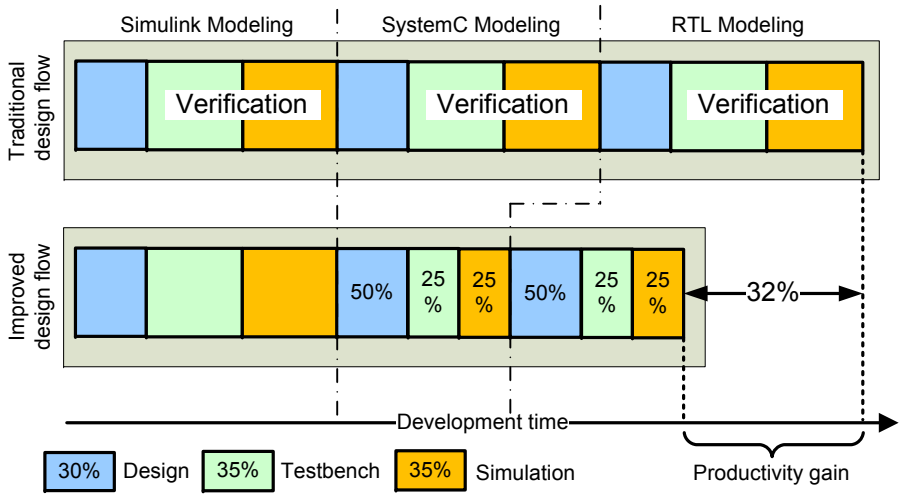

Figure 5–13: Vertical Testbench Reuse Productivity Gain

design can be simulated quickly with real world test scenarios; compared to other verification methods that required some time to create the testbench environment and test cases. Combining segmented adapters, SystemC verification library and Matlab/Simulink we created a unified verification framework that cuts verification time and increases the quality of functional verification.

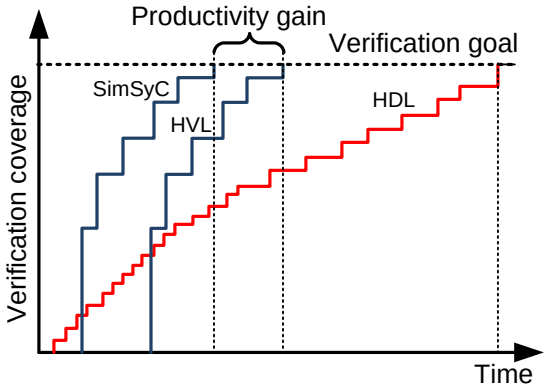

Figure 5–14: Verification productivity gain using SimSyC

Chapter 6
Conclusions and Future Work

This thesis studies a systematic approach to validate and verify complex digital system designs. The key contributions are summarized in section 6.1. In section 6.2 we conclude with a discussion of some of the future directions of this research.

6.1 Contributions

In this thesis we developed mechanisms and a platform for the verification of complex digital systems. With today's complex designs, the verification task has become the primary bottleneck in the design flow. The verification challenge is growing at a double exponential rate; that is, exponential with respect to Moore's law. A big part of verification is simulation. In spite of many research activities in design verification methodology, verification-by-simulation is still the major approach for its simple and intuitive way to address functional behavior of a hardware design. However, with multi-million gate digital systems, simulation requires complex testbenches. Manually coding all possible verification scenarios quickly becomes a fastidious and error prone task.

Our approach is to start the verification early in the design flow and to promote vertical testbench reuse across abstraction levels. Validation and verification are done in parallel to design with feedback and interaction between the two as the design progresses. The verification approach that we developed in this thesis has proven to be a valuable addition to the range of simulation-based methods already

available. The case study presented have shown the potential and main benefits of our contribution. However, the application domain of our platform is not restricted to these specific kind of circuits. It can be used for all kind of complex digital design that are represented at multiple levels of abstraction.

Four key challenges are identified [29] in the context of functional verification of digital designs: productivity, efficiency, reusability and completeness. We provide new and innovative solutions to the three former problems and these solutions are summarized in the next sections.

6.1.1 Segmented Adapter

The usage of transactors as presented in our verification methodology is an original contribution of this thesis. Transaction-based verification (TBV) uses transactors as an interface between the testbench and the design under verification. TBV raises the level of abstraction from signals to transactions, thus easing the development of reusable testbenches. However, with modern design flow starting at higher levels of abstraction, transactors need to be rewritten as the design is refined across abstraction levels. Time consuming transactor development is obviously making TBV less attractive in a multi abstraction level validation and verification environment. This problem was addressed in this thesis. We proposed a solution which generalizes the transactor concept. Segmented adapter are partitioned into three specific segments providing the modularity necessary for reuse and migration across abstraction levels and projects.

Our contribution has four main advantages:

- Transactor development time is reduced from project to project since only some segments need to be changed.
- The same segmented adapter can be used as the design is refined from one abstraction level to another. Only minor modifications need to be done with the abstraction translation segment and the design interface segment.
- The same segmented adapter can be used for designs represented in different design languages. Only the design interface segment needs to be changed.
- Modularity makes the maintenance and debugging of the segmented adapter much faster than traditional transactors.

6.1.2 SimSyc Cosimulation Interface

MATLAB/Simulink and SystemC are widely used for the design and verification of digital systems. Algorithmic models are created and optimized with Simulink while SystemC can be used for system-level modeling or as a functional verification language. To make the validation and verification environment truly reusable and scalable, the gap that exists between the algorithmic level and lower abstraction levels had to be bound. Hence, we proposed a co-simulation interface between SystemC and Simulink, namely SimSyC.

Using the SimSyC co-simulation interface we presented a method for reducing the time spent on validation and verification while improving overall testbench quality. MATLAB/Simulink assists the SystemC verification environment in a unified approach. It has been shown that SimSyC allows complex stimulus generation and exhaustive data analysis for the design under verification. As SoC designs encompass larger and larger systems, the need to efficiently model the complex external

environment during the architecture and verification phases becomes greater. Moreover, SimSyC is extremely valuable for SystemC users as it allows visualization of SystemC data through MATLAB/Simulink graphical toolboxes.

One last benefit of using SimSyC in the verification flow is for golden reference. A Simulink golden model can be used as a reference model by the verification system to compare the expected behaviours. Experiments show that our approach is able to validate and verify a SystemC design accurately and quickly using a golden reference.

6.1.3 Multi-Abstraction Levels

In order to solve the verification problem efficiently, it is desirable to create a methodology where the verification process starts as early as possible while providing the flexibility to move across abstraction levels. Such a requirement implies that multiple software tools and modeling languages will need to exchange data with the verification system. We proposed to use the SystemC modeling language with the verification library SCV to build the verification platform. Different from any other existing verification framework, our platform includes segmented interfaces and a verification backbone. The former link designs are represented at multiple levels of abstraction to the verification backbone. The backbone is the central element of the verification system where the testbench is created. Stimulus generation, response checking and the SimSyC co-simulation interface are all sub-components of this backbone.

6.2 Future Directions

The work in this dissertation can be extended in a number of areas. The SimSyC interface can be integrated in Simulink as a toolbox to facilitate its manipulation and

integration in various projects. A graphical user interface (GUI) is also required on top of the C-MEX S-Function, so it can be configured efficiently according to the design requirements. It would be interesting to investigate and incorporate a UNIX version of SimSyC. On Unix, MATLAB and Simulink do not have to run on the same computer as SystemC. This opens up the possibility of parallel processing between the data generator/analysis model and the design. Simulation performances can be substantially improved for application that requires computing intensive data generation models. Regarding the segmented adapter mechanism, the possible future research includes the development of a generic SystemC model that can be automatically configured according to design specifications. The process is actually performed manually and can be the source of errors. Another avenue would be to extend the verification platform with a hardware-in-the-loop interface to complete the link between abstraction levels. With this hardware interface, the testbench will be reused for in-circuit validation and verification. Finally, it would be interesting to try the verification platform with other data processing intensive applications that require complex data generation environment like video processing.

References

[1] M.A. Al-Qutayri, H.R. Barada, and A. Al-Kindi. Comparison of multiplier architectures through emulation and Handle-c FPGA implementation. In *Computer Systems and Applications, 2006. IEEE International Conference on.*, pages 240–247, March 8, 2006.

[2] K. Ara and K. Suzuki. A proposal for transaction-level verification with component wrapper language. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 82–87suppl., 2003.

[3] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC cosimulation and emulation of multiprocessor SoC designs. *Computer*, 36(4):53–59, April 2003.

[4] Janick Bergeron. *Writing Testbenches using SystemVerilog.* Springer, 2006.

[5] Janick Bergeron, Eduard Cerny, Alan Hunter, and Andy Nightingale. *Verification Methodology Manual for SystemVerilog.* Springer, 2005.

[6] Victor Berman. A tale of two languages: SystemC and SystemVerilog. In *Chip Design Magazine*. Extension Media, July 2005.

[7] T.M. Bhatt and D. McCain. Matlab as a development environment for FPGA design. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 607–610, 13-17 June 2005.

[8] F. Bouchhima, M. Briere, G. Nicolescu, M. Abid, and E.M. Aboulhamid. A SystemC/Simulink co-simulation framework for continuous/discrete-events simulation. In *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, pages 1–6, 2006.

[9] Dhananjay S. Brahme, Steven Cox, Jim Gallo1, Mark Glasser, William Grundmann, C. Norris Ip, William Paulsen, John L. Pierce, John Rose, Dean Shea, and Karl Whiting. The transaction-based verification methodology. Technical Report CDNL-TR-2000-0825, Cadence Berkeley Labs, August 2000.

[10] Inc. Cadence Design Systems. The unified verification methodology. http://www.cadence.com/whitepapers, 2005.

[11] N. Calazans, E. Moreno, F. Hessel, V. Rosa, F. Moraes, and E. Carara. From VHDL register transfer level to systemc transaction level modeling: a comparative case study. In *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, pages 355–360, 2003.

[12] F. Czerner and J. Zellmann. Modeling cycle-accurate hardware with matlab/simulink using systemc. 6th European SystemC Users Group Meeting (ESCUG), October 2002.

[13] K. Datta and P.P. Das. Assertion based verification using hdvl. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 319–325, 2004.

[14] Design and Reuse. Esl design corner, 2005.

[15] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.

[16] L. Drucker. Verification metrics - how you know when you're done. *Electronics Systems and Software*, 1(2):22–25, 2003.

[17] A. Fin, F. Fummi, and D. Signoretto. The use of systemc for design verification and integration test of ip-cores. In *ASIC/SOC Conference, 2001. Proceedings. 14th Annual IEEE International*, pages 76–80, 2001.

[18] Tom Fitzpatrick. Functional verification technology and methodology backgrounder. Mentor Graphics, 2005.

[19] D. Flynn. Amba: enabling reusable on-chip designs. *Micro, IEEE*, 17(4):20–27, 1997.

[20] Steve Forde, Steve Bishop, and Ramnath S. Velu. Streamlining hdl code coverage analysis. Integrated Systems Design, December 1998.

[21] R. Forster. Manchester encoding: opposing definitions resolved. *Engineering Science and Education Journal*, 9(6):278–280, 2000.

[22] P. Gerin, Sungjoo Yoo, G. Nicolescu, and A.A. Jerraya. Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 63–68, 30 Jan.-2 Feb. 2001.

[23] A. Gersho and T. Lim. Adaptative cancellation of intersymbol interference for data transmission. *Bell System Technical Journal*, 60:1997–2021, November 1981.

[24] Richard Goering. Dai introduces test-generation tool. EE Times, November 1998.

[25] Richard Goering. Intelligent testbenches gaining ground. EE Times, August 1999.

[26] Yuanbin Guo and D. McCain. Rapid prototyping and VLSI exploration for 3g/4g MIMO wireless systems using integrated Catapult-c methodology. In *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, volume 2, pages 958–963, 3-6 April 2006.

[27] Faisal Haque, Jonathan Michelson, and Khizar Khan. *The Art of Verification with Vera*. Verification Central, 2001.

[28] R. Hofmann and B. Drerup. Next generation coreconnect/spl trade/ processor local bus architecture. In *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, pages 221–225, 2002.

[29] Sasan Iman and Sunita Joshi. *The e hardware verification language*. Kluwer Academic Publishers, 2004.

[30] Peet James. *Verification plans: the five-day verification strategy for modern-hardware verification languages*. Kluwer Academic Publishers, 2004.

[31] J. Labat, O. Macchi, and C. Laot. Adaptive decision feedback equalization: can you skip the training period? In *IEEE Trans. Commun*, volume 46, pages 921–930, July 1998.

[32] William K. Lam. *Hardware Design Verification*. Pearson Education, Inc., 2005.

[33] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook*, volume 2. CRC, 2006.

[34] G. Martin. Systemc: from language to applications, from tools to methodologies. In *Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on*, page 3, 8-11 Sept. 2003.

[35] G. Martin. Verification by the pound. *Design & Test of Computers, IEEE*, 22(5):478–479, Sept.-Oct. 2005.

[36] Grant Martin. Systemc's role in a multilingual world. 8th European SystemC Users Group, November 2003.

[37] The MathWorks. Link for modelsim user's guide. http://www.mathworks.com/, December 2004.

[38] The MathWorks. Matlab and simulink, 2005.

[39] The MathWorks. *Link for ModelSim 2.0*, 2006.

[40] Clive Maxfield and Kuhoo Goyal Edson. *EDA: Where Electronics Begins*. TechBites Interactive, 2001.

[41] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 44(5):26–38, May 1995.

[42] Gordon E. Moore. Cramming more components onto integrated circuits. In *Electronics Magazine*, volume 38, pages pp. 114–117. McGraw-Hill, April 1965.

[43] A. Naumann. Esl - the next leadership opportunity for india? In *VLSI Design, 2005. 18th International Conference on*, page 26, 2005.

[44] Members of the SystemC Verification Working Group. *SystemC Verification Standard Specification*. OSCI, 1.0e edition, May 2003.

[45] Sanggyu Park and Soo-Ik Chae. A c/c++-based functional verification framework using the systemc verification library. In *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pages 237–239, 8-10 June 2005.

[46] David Pellerin and Douglas Taylor. *VHDL made easy!* Pearson Education Inc., 1997.

[47] C. Pixley, A. Chittor, F. Meyer, S. McMaster, and D. Benua. Functional verification 2003: technology, tools and methodology. In *ASIC, 2003. Proceedings. 5th International Conference on*, volume 1, pages 1–5Vol.1, 21-24 Oct. 2003.

[48] Andrew Piziali. *Functional verification coverage measurement and analysis*. Kluwer Academic Publishers, 2004.

[49] J. Proakis. *Digital Communications*. McGraw-Hill, 1995.

[50] MAME Project. Methodology and architecture of a multi-equalizer. http://www.ele.etsmtl.ca/projets/PROMPT/, June 2006.

[51] S. Qureshi. Adaptive equalization. In *Proceeding of the IEEE*, volume 73, pages 1349–1387, 1985.

[52] Collett International Research. 2005 ic/asic design closure study. Technical report, Collett International Research, 2005.

[53] A. Rose. System verification comes to systemc. Wireless Systems Design, September 2003.

[54] Sanghamitra Roy and Prith Banerjee. An algorithm for trading off quantization error with hardware resources for matlab-based fpga design. *Computers, IEEE Transactions on*, 54(7):886–896, July 2005.

[55] A. Sagahyroon, G. Lakkaraju, and M. Karunaratne. A functional verification environment. In *Circuits and Systems, 2005. 48th Midwest Symposium on*, pages 108–111Vol.1, 7-10 Aug. 2005.

[56] J. Salz. Optimum mean-square decision feedback equalization. *Bell System Technical Journal*, 8:1341–1373, October 1973.

[57] Michael Santarini. Cadence moves toward intelligent testbench. EE Times, June 1999.

[58] A. Sayinta, G. Canverdi, M. Pauwels, A. Alshawa, and W. Dehaene. A mixed abstraction level co-simulation case study using systemc for system on chip verification. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 95–100suppl., 2003.

[59] Rindert Schutten and Tom Fitzpatrick. Design for verification. Synopsys, Inc., 2003.

[60] E. Segev, S. Goldshlager, H. Miller, O. Shua, O. Sher, and S. Greenberg. Evaluating and comparing simulation verification vs. formal verification approach on block level design. In *Electronics, Circuits and Systems, 2004. ICECS 2004.*

Proceedings of the 2004 11th IEEE International Conference on, pages 515–518, 13-15 Dec. 2004.

[61] SIA. International technology roadmap for semiconductors. Technical report, ITRS, http://public.itrs.net, 2005.

[62] Gary Smith. The dream - communications/core-based design. Integrated System Design Magazine, December 2000. EETimes news.

[63] Ann Steffora. Dai enters transaction-based verification market. Electronic News, November 1998.

[64] S. Swan. Systemc transaction level models and rtl verification. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 90–92, 24-28 July 2006.

[65] M. Thompson, A.D. Pimentel, S. Polstra, and C. Erbas. A mixed-level cosimulation method for system-level design space exploration. In *Embedded Systems for Real Time Multimedia, Proceedings of the 2006 IEEE/ACM/IFIP Workshop on*, pages 27–32, Oct. 2006.

[66] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3):28–37, March 2007.

[67] Allen B Tucker and Robert Noonan. *Programming Languages: Principles and Paradigms*. McGraw-Hill Science, 2001.

[68] Inc. Verisity Design. A promising approach to overcome the verification gap of modern soc designs. http://www.verisity.com, 2004.

[69] C. Warwick. Systemc calls matlab. MATLAB Central, March 2003.

[70] Lin Yi-Fan, Zeng Xiao-Yang, Wu Min, Chen Jun, and Bao Rencheng. New methods of fpga co-verification for system on chip (soc). In *ASIC, 2005. ASICON 2005. 6th International Conference On*, volume 1, pages 219–222, 24-27 Oct. 2005.

[71] S. Yoo and A.A. Jerraya. Hardware/software cosimulation from interface perspective. In *Computers and Digital Techniques, IEE Proceedings-*, volume 152, pages 369–379, 6 May 2005.