System Design for DSP Applications with the MASIC Methodology

Abhijit Kumar Deb

Doctoral Thesis
Stockholm, Sweden 2004

Other Publications

Licentiate Theses:

Johnny Öberg	An Adaptable Environment for Improved High-level Synthesis, 1996.
Bengt Oelmann	Design and Performance Evaluation of Asynchrounus Micropipline Circuits for Digital Radio, 1996.
Henrik Olson	ASIC Implementable Synchronization and Detection Methods for Direct Sequence Spread Spectrum Wideband Radio Receivers, 1997.
Mattias O'Nils	Hardware/Software Partitioning of Embedded Computer Systems,1996.
Daniel Kerek	Design of A Wideband Direct Sequence Spread Spectrum Radio Transceiver ASIC, 1999.
Bengt Svantesson	Dynamic Process Management in SDL to VHDL Synthesis, 2000.
Johnny Holmberg	On Design and Analysis of LDI and LDD Lattice Filters, 2000.
Bingxin Li	Design of Sigma Delta Modulators for Digital Wireless Communications, 2001.
Chuansu Chen	Reusable Macro Based Synthesis for Digital ASIC Design, 2002.
Li Li	Noise Analysis of Mixers for RF Receivers, 2002.
Tomas Bengtsson	Boolean Decomposition in Combinational Logic Synthesis, 2003.
Meigen Shen	Chip and Package Co-Design for Mixed-Signal Systems: SoC versus SoP, 2004.
Jimson Mathew	Design and Evaluation of Fault Tolerant VLSI Architectures, 2004.
Petra Färm	Advanced Algorithms for Logic Synthesis, 2004.
René Krenz	Graph Dominators in Logic Synthesis and Verification, 2004.
Andrés Martinelli	Advanced Algorithms for Boolean Decomposition, 2004.
Tarvo Raudvere	Verification of Local Design Refinements in a System Design Methodology, 2004.

Doctoral Theses:

Tawfik Lazraq	Design Techniques and Structures for ATM Switches, 1995.
Bengt Jonsson	Switched-Current Circuits: from Building Blocks to Mixed Analog-Digital Systems, 1999.
Johnny Öberg	ProGram: A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols, 1999.
Mattias O'Nils	Specification, Synthesis and Validation of Hardware/Software Interfaces, 1999.
Peeter Ellervee	High-Level Synthesis of Control and Memory Intensive Applications, 2000.
Henrik Olson	Algorithm-to-Architecture Refinement for Digital Baseband Radio Receivers, 2000.
Bengt Oelmann	Asynchronous and Mixed Synchronous/Asynchronous Design Techniques for Low Power, 2000.
Yonghong Gao	Architecture and Implementation of Comb Filters and Digital Modulators for Oversupplying A/D and D/A Converters, 2001.
Lirong Zheng	Design, Analysis and Integration of Mixed-Signal Systems for Signal and Power Integrity, 2001.
Per Bjuréus	High-Level Modeling and Evaluation of Embedded Real-Time Systems, 2002.
Imed Ben Dhaou	Low Power Design Techniques for Deep Submicron Technology with Application to Wireless Transceiver Design, 2002.
Ingo Sander	System Modeling and Design Refinement in ForSyDe, 2003.
Andreas Göthenberg	Modeling and Analysis of Wideband Sigma-Delta Noise Shapers, 2003.
Dinesh Pamunuwa	Modeling and Analysis of Interconnects for Deep Submicron Systems-on-Chip, 2003.
Bingxin Li	Design of Multi-bit Sigma-Delta Modulators for Digital Wireless Communications, 2003.
Li Li	Modeling, Analysis and Design of RF Mixed-Signal Mixer for Wireless Communication, 2004

System Design for DSP Applications with the MASIC Methodology

Abhijit Kumar Deb

ROYAL INSTITUTE
OF TECHNOLOGY

Stockholm 2004

Laboratory of Electronics and Computer Systems
Department of Microelectronics and Information Technology
Royal Institute of Technology, Stockholm, Sweden

Thesis submitted to the Royal Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Technology

Deb, Abhijit K.
System Design for DSP Applications with the MASIC Methodology

TRITA-IMIT-LECS AVH 04:10
ISSN 1651-4076
ISRN KTH/IMIT/LECS/AVH-04/10--SE

Royal Institute of Technology
Department of Microelectronics and Information Technology
Laboratory of Electronics and Computer Systems
Isafjordsgatan 39
S-164 40 Kista, Stockholm, Sweden

Abstract

The difficulties of system design are persistently increasing due to the integration of more functionality on a system, time-to-market pressure, productivity gap, and performance requirements. To address the system design problems, design methodologies build system models at higher abstraction level. However, the design task to map an abstract functional model on a system architecture is nontrivial because the architecture contains a wide variety of system components and interconnection topology, and a given functionality can be realized in various ways depending on cost-performance tradeoffs. Therefore, a system design methodology must provide adequate design steps to map the abstract functionality on a detailed architecture.

MASIC—*Maths to ASIC*—is a system design methodology targeting DSP applications. In MASIC, we begin with a functional model of the system. Next, the architectural decisions are captured to map the functionality on the system architecture. We present a systematic approach to classify the architectural decisions in two categories: *system level decisions* (SLDs) and *implementation level decisions* (ILDs). As a result of this categorization, we only need to consider a subset of the decisions at once. To capture these decisions in an abstract way, we present three *transaction level models* (TLMs) in the context of DSP systems. These TLMs capture the design decisions using abstract transactions where timing is modeled only to describe the major synchronization events. As a result the functionality can be mapped to the system architecture without meticulous details. Also, the artifacts of the design decisions in terms of delay can be simulated quickly. Thus the MASIC approach saves both modeling and simulation time. It also facilitates the reuse of predesigned hardware and software components.

To capture and inject the architectural decisions efficiently, we present the grammar based language of MASIC. This language effectively helps us to implement the steps pertaining to the methodology. A Petri net based simulation technique is developed, which avoids the need to compile the MASIC description to VHDL for the sake of simulation. We also present a divide and conquer based approach to verify the MASIC model of a system.

Keywords: System design methodology, Signal processing systems, Design decision, Communication, Computation, Model development, Transaction level model, System design language, Grammar, MASIC.

To my dear parents and caring wife

Acknowledgements

I would like to take the opportunity to express my sincere gratitude to my academic advisors, Dr. Johnny Öberg and Prof. Axel Jantsch for their support and encouragement over the last couple of years. I always felt myself lucky to have them as my academic advisors.

Dr. J. Öberg has been deeply involved with this PhD project since its commencement, and helped me in many different ways. His ingenious ideas toward the improvement of the project, invaluable suggestions with the grammar based language used in this project, and moral support have been the constant source of motivation throughout my PhD studies.

Prof. A. Jantsch has played the important part in introducing and explaining to me the role of different models of computation for modeling concurrency and time in a system model. No research project can keep pace with the rapidly evolving technology without meticulous study of contemporary research activities. He has always inspired me to read more and more research papers, and instigated me to work more by setting a high target.

I am especially thankful to Prof. Hannu Tenhunen who gave me the opportunity to start with this PhD project. Without him I would not even be at KTH and my work and contributions, whatever they are worth, would be very different, most likely less interesting. I am also very much thankful to Dr. A. Hemani for his genuine support and supervision during the early phase of this project. He made it possible for me to make a quick start with the project.

My colleagues and fellow PhD students have helped me through many mind-boggling discussions and numerous acts of friendship. In particular, I am thankful to Ingo Sander for many priceless discussions on how to write a technical paper; to Steffen Albrecht for being a real friend; to Dinesh Pamunuwa for joining and equally “giving-miss” to too many coffee breaks; to Andreas Göthenberg for all the fuzzy discussions; and to Wim Michielsen for all his “creative sketches” on my white board.

During the course of this work, many others in this lab have extended their helping hand. In particular, I would like to acknowledge the sincere support in many different ways that I have got from Dr. Elena Dubrova and Dr. Li-Rong Zheng.

I would like to acknowledge the system support group, especially Hans Berggren, for their system support. I would also like to acknowledge Lena Beronius and Margreth Hellberg, who helped me in various ways – starting from getting an accommodation in Stockholm, to several administrative helps.

This work has been funded by the Swedish Foundation for Strategic Research – SSF, Intelect, DOCCS, and KTH.

PhD studies cannot go on without special sacrifices of the family members, and this is no exception. Finally, I would like to acknowledge my biggest debt to my dear parents for their constant support and encouragement, and to my caring wife for her love and affection. I deprived her of many weekends and evenings while chasing for different paper submission deadlines.

Stockholm, September 2004
Abhijit K. Deb

Contents

List of Abbreviations

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASB	Advanced System Bus
ASIC	Application Specific Integrated Circuit
BFM	Bus Functional Model
BNF	Backus-Naur Form
BTM	Bus Transaction Model
CFSM	Codesign Finite State Machine
CAD	Computer Aided Design
CSP	Communicating Sequential Process
DAG	Directed Acyclic Graph
DDF	Dynamic Data Flow
DFG	Data Flow Graph
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FIFO	First In First Out
FIR	Finite Impulse Response
FM	Functional Model
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FSMD	Finite State Machine with Datapath
GLOCCT	GLObal Control, Configuration, and Timing
HDL	Hardware Description Language
I/O	Input/Output
IC	Integrated Circuit
ILD	Implementation Level Decision
IP	Intellectual Property
IM	Implementation Model
ITRS	International Technology Roadmap for Semiconductors
KPN	Kahn Process Network
MASIC	Maths to ASIC
MPSOC	Multiprocessor System-on-Chip
NOC	Network-on-Chip
NRE	Non-recurring Engineering
PTM	Process Transaction Model

RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
RTOS	Real Time Operating System
SDF	Synchronous Data Flow
SFG	Signal Flow Graph
SLD	System Level Decision
SOC	System-on-Chip
STM	System Transaction Model
TLM	Transaction Level Model
VC	Virtual Component
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Large Instruction Word
VLSI	Very Large Scale Integration
YACC	Yet Another Compiler Compiler

1. INTRODUCTION

This chapter provides an introduction to this dissertation. We start with a short background of the work. Next, we describe the motivation behind the development of the MASIC methodology, and describe the methodology in brief. Finally, we point out the contribution of the author in the development of the methodology followed by a short description on the organization of this dissertation.

1.1 Thesis Background

Almost all aspects of our way of life have been heavily influenced by the continued proliferation of electronic systems. The design task of these electronic systems encompasses a wide area of knowledge and research activities. This chapter begins with a look on the backdrop of the work presented in this dissertation so that the reader can put it in the right context within the vast field of electronic systems design.

1.1.1 Historical Perspective

As far back as 1965, Gordon Moore[1] made the classical observation that the number of transistors per integrated circuit was exponentially increasing with time, and predicted that this trend would continue [104]. His insightful empirical law, which became known as *Moore's law*, still holds true and the exponential growth is expected to continue at least through the end of this decade.

[1] He was the director of R&D of Fairchild Semiconductor—the company that he co-founded—during the time he wrote his legendary paper, and joined Intel Corporation afterwards. In his paper, he predicted that the exponential growth of semiconductor circuits would continue at least for the next ten years, and most likely to continue beyond that time.

It is the relentless efforts of researchers and engineers that have made it possible to sustain such an exponential growth over the last four decades. As a result we have experienced that the number of devices that could be economically fabricated on a single chip has been exponentially increasing at a rate of 50% per year and hence quadrupling every 3.5 years. The delay of a simple gate has been decreasing by 13% per year, halving every 5 years [34]. While the minimum feature size is decreasing, the chip size is increasing. The million-transistor/chip barrier was crossed in the late 1980s. The clock frequency doubles every three years [112]. A graph of the enormous pace of the growth of integration density of microprocessors and memories is shown in Figure 1.

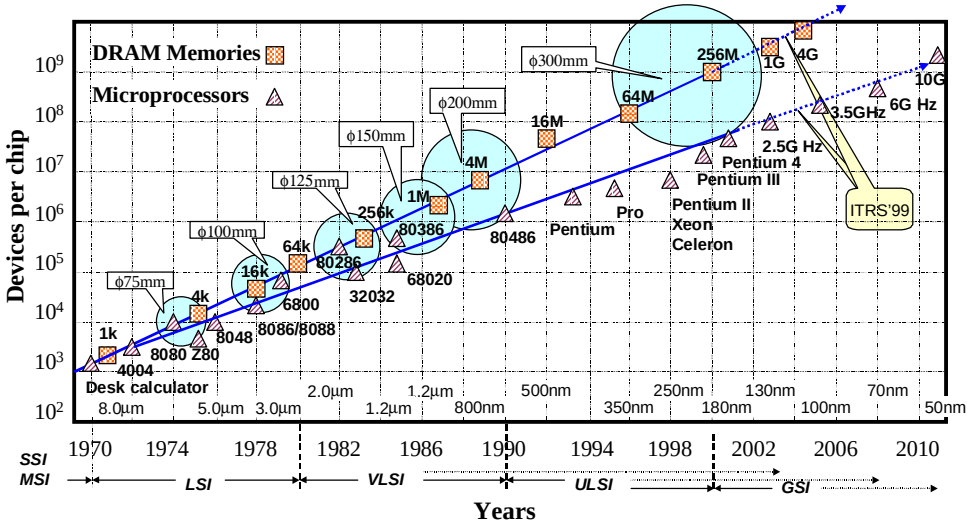

Figure 1. Evolution of microelectronics (after [136])

The increase in the size, speed and functionality of VLSI circuits in turn increased the complexities of designing and fabricating such circuits. To manage the complexities, it became customary to decouple the design engineer from requiring to understand the intricacies of the fabrication process. Therefore, a set of *design rules* was defined that acts as the interface or even the contract between the design engineer and the process engineer [112]. Design engineers generally want tighter and smaller designs, which lead to higher performance. The process engineers, on the other hand, want a reproducible and high-yield process. Consequently, the design rules are a compromise that attempts to satisfy both sides. The design rules provide a set of guidelines for constructing the various masks needed in the

patterning process. They consist of minimum-width and minimum-spacing constraints, and the requirements between objects on the same or different layers.

1.1.2 Barriers to Moore's Law

The exponentially growing availability of transistors makes it plausible to build larger systems, which integrate more functionality on a single die. The inevitable consequence of this enormous scale of integration increases the complexity of the system, and makes it more challenging to cost-effectively design and verify them.

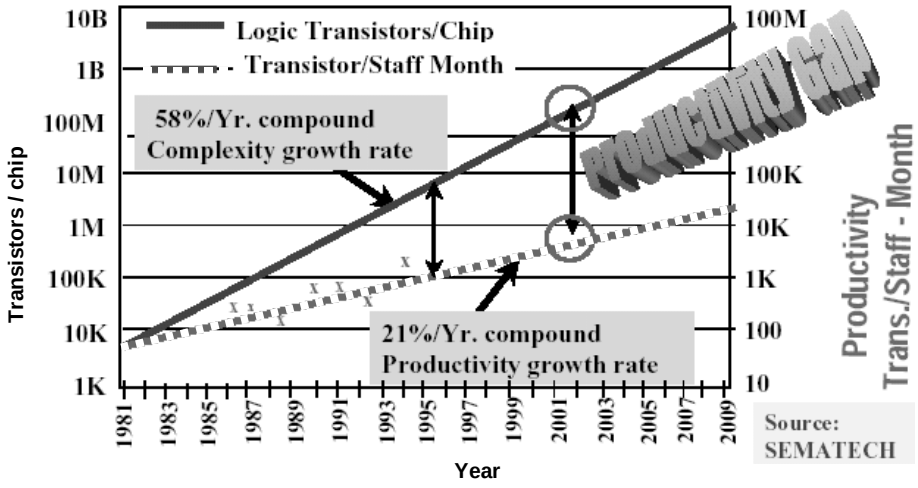

Figure 2. The increasing design productivity gap (after [136])

Over the years we have observed a discrepancy between the pace at which the design technology and the process technology advances. The International Technology Roadmap for Semiconductors (ITRS) has documented a *design productivity gap* in which the number of available transistors grows faster than the ability to meaningfully design them. Figure 2 shows the increasingly widening gap between the manufacturing capability and the design productivity that we are experiencing over time.

The main message in the 2003 ITRS roadmap is that the cost of design is the greatest threat to the continuation of the semiconductor industry's phenomenal growth [4]. Manufacturing non-recurring engineering (NRE) costs are in the order

of one million dollars (mask set and probe card), whereas design NRE costs routinely reach tens of millions of dollars. When manufacturing cycle times are measured in weeks, with low uncertainty; the design and verification cycle times are measured in months or years, with high uncertainty [12]. However, investment in process technology has far outstripped the investment in design technology [49]. Nonetheless, progresses in the design technology are evolving. The work presented in this dissertation focuses on one aspect of design technology—system level design—with the aim to increase the design productivity.

1.1.3 Systems and Models

Arguably, almost everyone has a different definition of what a system is. In this context, a nice discussion can be found in [76], "What is meant by a system always depends on the context of discourse. What is a system in one context may be a tiny component in another context and the environment in a third." For instance, an Application Specific Integrated Circuit (ASIC) is a very complex system to ASIC designers, but it is only one out of many electronic components that goes into an airplane, which in turn is only one element in the air traffic organization of a country. In this thesis, by "system" we mean electronic systems, such as those built into mobile phones, base-stations, digital set-top boxes, digital cameras, personal digital assistants, cars, planes, toys, robots, and so on. Signal processing remains at the heart of most of these products. Often, these products include an embedded microprocessor, if not more, to implement functionalities in software, and they are commonly referred to as embedded systems[2].

A modern car is known to contain up to 70 separate embedded controllers [127]. The March 2004 edition of *IEEE Spectrum* features a cover story on "Top 10 Tech Cars" [130]. This article describes the latest developments of automobile electronic systems like *roll stability control*, *active steering system*, *lane keeping assistant system*, etc. These systems acquire data from steering, braking, and vehicle sensors that monitor speed, body roll, yaw, wheel angle, etc. Then they perform a lot of signal processing on the sensor data and images from windshield mounted camera, and finally controls the brake torque at specific wheels, the front wheel angle, the engine power, the air bags, etc. According to Daimler-Chrysler sources, more than 90% of the innovation (and hence the value added) in a modern car is in electronics [118]. According to BMW, electronic components comprise more than 30% of a car's manufacturing cost. Considering these data, the author

[2] An embedded system is defined as a system, which uses a computer to perform a specific function, but is neither used nor perceived as a computer [50].

has very reasonably put forward the question: “Will the electronic components be the car and the mechanical components an accessory?”

Although microprocessors has been in use for over two decades, microprocessor based systems have almost exclusively been board-level systems. By the 1990s, microprocessor based systems became an important design discipline for integrated circuit (IC) designers [134]. As a result, designers opted for realizing the whole microprocessor based system within a System-on-Chip (SOC). Since then, even multiprocessor System-on-Chip (MPSOC) has started to enter the marketplace, and are expected to be available in even greater variety over the next few years [135]. MPSOC solutions make most sense in high volume markets that have strict performance, power, and cost requirements. Communications, multimedia, and networking are the market segments that meet these requirements. As a result, we see the emergence of different MPSOC designs, like the Philips Nexperia for digital set-top boxes and digital TV systems, the ST Nomadik for mobile multimedia applications, the TI OMAP for wireless applications, and the Intel IXP2850 for network processing [135].

Current electronic systems are far more complex than those of a decade ago. While personal digital assistants contain a large 32-bit CPU, the high-end cameras, digital set-top boxes and laser printers contain multiple processors [134]. Philips Nexperia architecture contains two processors–a 32-bit MIPS CPU and a TriMedia Very Large Instruction Word (VLIW), plus three buses and a number of other units to implement special purpose functions [46]. It decrypts, decodes, converts, and displays multiple media streams having different data formats. The system typically handles live video, audio, and various other stream types in compressed and uncompressed formats. The set top box designed at STMicroelectronics contains a proprietary 64-bit processor and some general purpose blocks, along with dedicated functions specifically designed by STMicroelectronics [113]. Chip complexity is in the range of 8 million gates, 3 Mbits of embedded static RAM, integration of analog IP’s[3], and 400 signal pads. The main clock frequency is 200 MHz and circuits are organized around an internal proprietary bus.

The scale of system integration in turn increases the design complexity. As a result, it becomes indispensable to devise some technique to manage the design complexity. There are two major ways of managing complexities as shown in Figure 3. Partitioning divides the complexities into smaller chunks, without reducing them. The concept of abstraction, on the other hand, reduces the

[3] In this text, we shall use the term *IP* to mean *Intellectual Property*. To avoid potential confusion with the more popular use of the term *IP* as *Internet Protocol*, some people refer to them as *Virtual Components* (VCs).

complexities by hiding details. Abstraction is the measure of absence of details. To be able to design and verify complex systems with ease, it has become customary to construct abstract models of them. A model is a simplification of another entity, which can be a physical thing or another model [76]. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task.

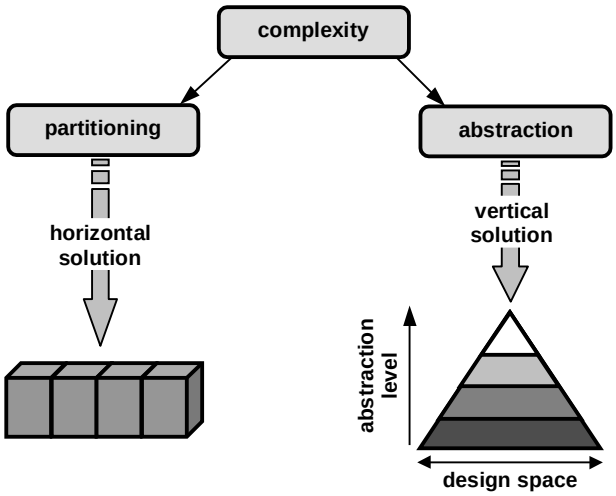

Figure 3. Two ways of managing design complexities

The higher the level of abstraction is, the less detailed the model is. Electronic Design Automation (EDA) techniques and tools are often classified by the level of abstraction they deal with, where system level design remains at the top of the hierarchy. This dissertation presents a particular way of modeling signal processing systems at higher levels of abstraction, and presents a system level design methodology.

1.1.4 System Level Design

As we build system models at higher levels of abstraction, the modeling and verification tasks become easier. This is because the abstract model relieves the designer to think about the rigorous implementation details at the beginning, and gives the freedom to explore the design space with different alternatives per se. Then, the next question is how to realize a detailed implementation of the abstract model. The act of design adds more details to the abstract model and yields an

implementation model, most likely through a number of intermediate levels of abstraction, as depicted in Figure 4.

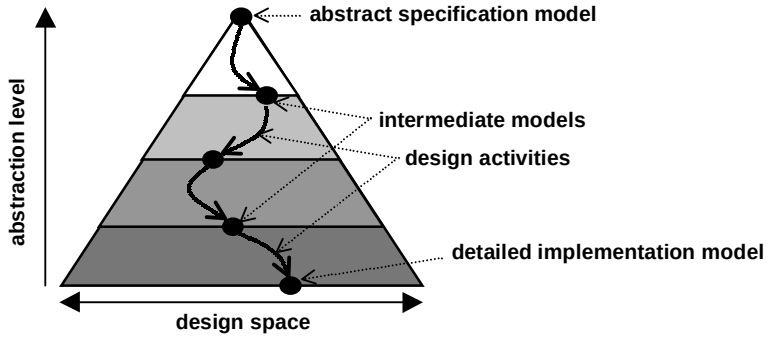

Figure 4. Design process from higher abstraction level to implementation level

System level design deals with realizing the system functionality on a system architecture. The system functionality is captured using different abstract formalisms, which we shall discuss in chapter 2. The overall system architecture includes the integration of different system components like general purpose processor cores[4], digital signal processing (DSP) cores, and VLIW cores to achieve flexibility and programmability; custom hardware blocks to implement performance critical portions; predesigned hardware blocks to save design time; and memory subsystems for the embedded processors. Moreover, to provide application specific reconfigurability, it will also be required to have uncommitted Field Programmable Gate Array (FPGA) circuitry in the system. To connect these system components we need some kind of communication architecture, either bus based or network based. In addition, the SOC architecture would also contain I/Os, data converters, and mixed analog/digital designs. In this dissertation we focus on the digital part of the SOC architecture, targeting signal processing applications, and present a design methodology that facilitate the designer to map an abstract functional model to a system architecture.

[4] An integrated circuit *core* is a predesigned, preverified silicon circuit block, usually containing at least 5000 gates that can be used in building a larger or more complex application on a semiconductor chip [64]. A core may be soft, firm, or hard. A soft core consists of a synthesizable HDL description that can be integrated to different semiconductor processes. A firm core contains more structure, commonly a gate-level netlist that is ready for placement and routing. A hard core includes layout and technology dependent timing information, and is ready to be dropped into a system.

1.2 Motivation

The pursuit to integrate more functionality and gain flexibility to support multiple standards, typical systems embrace heterogeneous architectures. The architecture is often comprised of a handful of programmable processor cores, different hardware blocks, few different segments of buses using different data transfer protocols, and bridges among these different bus protocols. The enormous scale of integration increases the system design complexity. On top of the scale of system integration, the designers face strict performance requirements, which in turn add up to the design challenges. To make the scenario even worse, the increasing time-to-market pressure demands these complex systems to be designed and verified in less amount of time. However, the established design methodologies in industry use the register transfer (RT) level description, which fail to cope with the design complexities due to a number of reasons:

- RT level is too detailed to be effective enough to explore the design space with different design alternatives.
- The system functionality often gets mixed up with the implementation details, which makes the design task more complex, and hinders design reuse and design for reuse.
- The hardware/software co-verification tools come into play at a later stage when the design team has already made much of the commitments to an implementation.
- There is no smooth path from the functional modeling phase to the implementation level. Often the architectural decisions are captured in plain natural language at higher levels of abstraction, and the results of the functional modeling phase is not effectively reused at the RT level.

To tackle these design challenges, we believe that there is a clear need to adopt a higher level of abstraction to model complex systems. The abstract system model needs to be free of rigorous implementation details and pave the way to implement it in various different ways. Mapping an abstract functional model on a system architecture is a nontrivial task because the architecture contains a wide variety of system components and interconnection topology, and a given functionality can be realized in various ways depending on cost-performance tradeoffs. Therefore, a system design methodology must provide adequate design steps, possibly through a number of levels of abstraction, to map the functionality on a system architecture. Along the way, the methodology must support the reuse of

predesigned hardware and software components, and provide us with as accurate performance estimates as possible.

MASIC—***Maths to ASIC***—is a system design methodology targeting DSP applications that tackles the system design challenges. The methodology has been introduced in [35][70][72]. The basic essence of the design flow in MASIC is the clean separation between the two major aspects of a system:

- *Functionality*: The functionality is the abstract behavior of the system that tells us what the system is supposed to do.

- *Architectural decisions*: These decisions tell us how the functionality is implemented, that is how the individual functional blocks perform their computation, and how these blocks communicate with each other and with the environment.

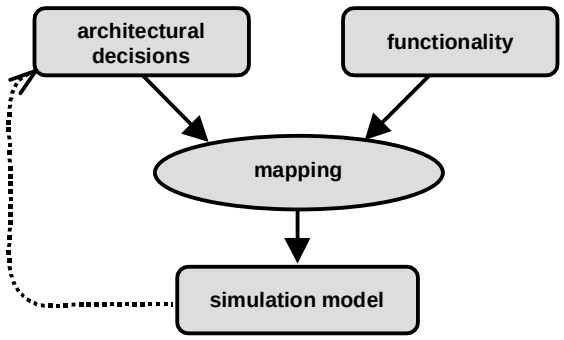

Figure 5. The basis of the system design flow with the MASIC methodology

The basis of the system design flow with MASIC is shown in Figure 5. In this design flow, the functionality is modeled separately without any concern of the implementation architecture. The architectural decisions capture the information about the implementation of the functionality. Finally, they are mapped together to validate that the given functionality runs well on the chosen architecture, and meets the performance requirements. The separation of the two design aspects splits the system design task into smaller and more manageable problems.

In the context of the separation of these design aspects, MASIC has a similar approach to the interface based design methodologies, which separate the system behavior from the implementation architecture [96][114][116]. Other contemporary methodologies, like the YAPI for signal processing system design

developed by Vissers et al. [44][84][131], the function architecture codesign developed by Sangiovanni-Vincentelli et al.[122], and the platform based design methodology developed by Keutzer et al. [83][102], also advocate the separation between the functionality and the architectural decisions.

To model a system using the design flow shown in Figure 5, a system designer typically studies the functionality, takes the system specification, makes a few initial calculations, and then proposes an architecture to implement the functionality. Next, the system performance is evaluated, for instance by simulation [84][131], and architectural decisions are iteratively altered to meet the performance goals. Although the concept of the separation of the functionality and the architectural decisions is of great value for any system design methodology per se, it brings three more issues that call for special treatment:

- There exists a huge number of architectural alternatives to implement both the computation and the communication aspects of a system. Exploring all the architectural decisions is an inefficient approach. Hence there is a need to categorize the set of architectural decisions in a systematic way, such that only a subset of the decisions needs to be considered at any particular instance in the design flow.

- There is a top-down iteration in the design flow shown in Figure 5. As a result, if a given set of architectural decisions does not yield satisfactory performance then the design cycle needs to be iterated for another set of decisions. Such a long top-down iteration hinders the design productivity and there is a clear need of creating intermediate abstraction levels such that the top-down iteration is broken into smaller loops.

- Exploring the design space with different architectural decisions becomes difficult if the decisions are captured at a lower abstraction level where the data transactions among the modules are described elaborately. Therefore, we need to map the functionality to the system architecture at higher abstraction level such that the system models can be created quickly, and simulated with faster speed. In addition, we also need the abstract models to yield as accurate performance estimates as possible.

The MASIC methodology addresses all the preceding issues. In the following section, we briefly discuss the key concepts of the system design flow in MASIC and the tool support.

1.3 Key Concepts of the MASIC Methodology

We present a systematic approach to classify the architectural decisions in two categories: ***system level decisions*** (SLDs) and ***implementation level decisions*** (ILDs) [41]. As a result of this categorization, we only need to consider a subset of the decisions at once. Figure 6 illustrates the conceptual design flow after categorizing the architectural decisions. We add the SLDs to the functional model to create an abstract intermediate model. Next, the ILDs are added to the intermediate model to build a more detailed model. Figure 6 clearly shows that the creation of the intermediate abstraction level breaks the top-down iteration of the initial design flow of Figure 5 into two smaller loops. Next, we capture the SLDs and ILDs in an abstract way.

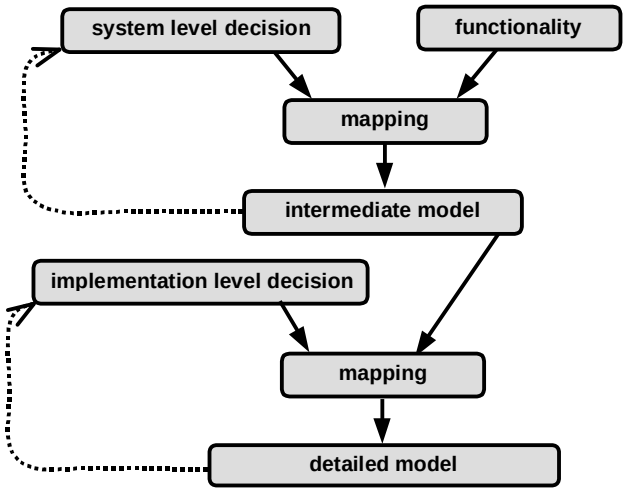

Figure 6. System design flow after categorization of architectural decisions

Typically, the ***functional model*** (FM) of a DSP system is built at the most abstract level where both communication and computation takes place in zero-time. In the ***implementation model*** (IM), however, both communication and computation are cycle accurate. Between the FM and IM of a system, we have created three ***transaction level models*** (TLMs) in the context of signal processing systems [42]. These TLMs are the ***process transaction model*** (PTM), the ***system transaction model*** (STM), and the ***bus transaction model*** (BTM). These TLMs capture the design decisions using abstract transactions where timing is modeled only to describe the major synchronization events. As a result the functionality can be mapped to the system architecture without meticulous details. Also, the artifacts of

the design decisions in terms of delay can be simulated quickly so that fast and reasonably accurate performance estimates can be obtained. Thus the MASIC approach saves both modeling and simulation time, and effectively helps the designer to explore the design space with alternative design decisions.

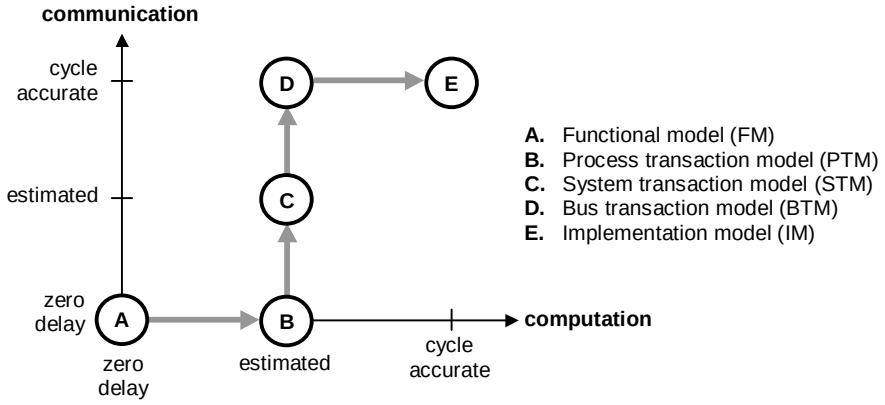

Figure 7. Coordinates of different system models in the system modeling graph

Gajski et al. have presented the *system modeling graph* where computation and communication are shown on two axes [28]. Each axis has three degrees of time accuracy: *untimed*, *estimated*, and *cycle accurate*. The coordinates of our different system models are shown in Figure 7. The thick arrow in gray shows the design flow in MASIC. Throughout the design flow, design decisions are added to an abstract model to create a less abstract model.

1.3.1 Language Support and Validation Technique

To successfully exploit the advantages offered by the MASIC methodology there is a need to have adequate tool support to build and simulate the system models. In our design flow, from the functional model through the transaction level models (TLMs) to the final implementation, the functionality remains the same. It is only the protocol of data transaction that evolves from abstract FIFO channels to bus protocols and component interfaces, like the *bus functional models* (BFMs) of embedded cores, etc. To build the system models at different abstraction levels, we need a suitable language to express the communication protocol. Grammars provide a natural way of describing communication protocol. Therefore, we have developed a grammar based language, which is built upon the principles of the

ProGram [140] language. Abstract communication modeling has become a part of different system modeling languages like SystemC [59] and SpecC [48]. However, we argue that grammars provide a more intuitive and natural way of describing protocols than the C++ or C language.

The grammar based language of MASIC provides a unique way of capturing the architectural decisions. It creates the system interface, the interfaces of the individual functional blocks, the data channels among the blocks and the data transactions within the channels. Signal processing algorithms are typically developed in a C or MATLAB-like environment. The MASIC compiler imports these C functions, and generates a high level description of the system model in VHDL. The foreign language interface (FLI) of VHDL is used to perform co-simulation between VHDL and C.

It is not possible to simulate the grammar descriptions of the system models, unless they are compiled to another language, for example VHDL. Therefore, to be able to simulate the grammar descriptions, we have developed a Petri net representation of the models. As a result, the MASIC models can be simulated without having to compile them to another language. The Petri net representation of the system model also paves the way to apply different Petri net based analysis techniques, like deadlock or reachability analyses, on the model.

Verification of complex systems is a difficult problem because many DSP blocks placed on a system require a complicated global control logic for their proper interaction. Verifying the DSP blocks together with the global control results in a complex validation problem. We propose a divide and conquer approach that takes advantage of the clean separation between the functionality and the architectural decisions in the MASIC model of a system, and speeds up the verification process substantially.

1.4 Publications

This dissertation is based on the following publications by the author. All the papers listed here are peer-reviewed:

[paper-i] Abhijit K. Deb, A. Jantsch, and J. Öberg, “System Design for DSP Applications in Transaction Level Modeling Paradigm,” in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, pp. 466-471, San Diego, CA, Jun. 2004.

[paper-ii] Abhijit K. Deb, A. Jantsch, and J. Öberg, “System Design for DSP Applications Using the MASIC Methodology,” in *Proc. IEEE Design,*

Automation and Test in Europe (DATE) Conf., vol. 1, pp. 630-635, Paris, France, Feb. 2004. (Nominated for best paper award in its category)

[paper-iii] Abhijit K. Deb, J. Öberg, and A. Jantsch, "Simulation and Analysis of Embedded DSP Systems using Petri Nets," in *Proc. 14th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pp. 64-70, San Diego, California, Jun. 2003.

[paper-iv] Abhijit K. Deb, J. Öberg, and A. Jantsch, "Simulation and Analysis of Embedded DSP Systems Using MASIC Methodology," in *Proc. IEEE Design, Automation and Test in Europe (DATE) Conf.*, pp. 1100-1101, Munich, Germany, Mar. 2003.

[paper-v] Abhijit K. Deb, A. Hemani, J. Öberg, A. Postula, and D. Lindqvist, "Hardware Software Codesign of DSP Systems using Grammar Based Approach," in *Proc. 14th IEEE Int. Conf. on VLSI Design*, pp. 42-47, Bangalore, India, Jan. 2001.

[paper-vi] Abhijit K. Deb, A. Hemani, and J. Öberg, "Grammar Based Design and Verification of an LPC Speech Codec: A Case Study," in *Proc. 11th Int. Conf. on Signal Processing Applications and Technology (ICSPAT'2000)*, Dallas, USA, Oct. 2000.

[paper-vii] A. Hemani, Abhijit K. Deb, J. Öberg, A. Postula, D. Lindqvist and B. Fjellborg, "System Level Virtual Prototyping of DSP SOCs Using Grammar Based Approach," *Kluwer Journal on Design Automation for Embedded Systems*, vol. 5, no. 3/4, pp. 295-311, Aug. 2000.

[paper-viii] Abhijit K. Deb, A. Hemani, and J. Öberg, "A Heterogeneous Modeling Environment of DSP Systems Using Grammar Based Approach," in *Proc. 3rd Int. Forum on Design Languages (FDL)*, pp. 365-370, Tubingen, Germany, Sep. 2000.

[paper-ix] Abhijit K. Deb, A. Hemani, A. Postula and L.S. Nagurney, "The Virtual Prototyping of an AM Chip Using Grammar Based Approach," in *Proc. 17th IEEE NORCHIP Conf.*, pp. 283-290, Oslo, Norway, Nov. 1999.

[paper-x] A. Hemani, A. Postula, Abhijit K. Deb, D. Lindqvist and B. Fjellborg, "A Divide and Conquer Approach to System Level Verification of DSP ASICs," in *Proc. IEEE Int. High Level Design Validation and Test Workshop (HLDVT)*, pp. 87-92, San Diego, California, Nov. 1999.

[paper-xi] A. Hemani, J. Öberg, Abhijit K. Deb, D. Lindqvist and B. Fjellborg, "System Level Virtual Prototyping of DSP ASICs Using Grammar Based Approach," in *Proc. 10th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pp. 166-171, Clearwater, Florida, Jun. 1999.

1.5 Author's Contribution

The author of this thesis started working with the MASIC project during the early 1999. That was roughly the time when the project was formulated by Dr. A. Hemani and Dr. J. Öberg. After Dr. A. Hemani had left the Royal Institute of Technology (KTH) for an industrial career, Prof. A. Jantsch became involved with the project. Since the commencement of the project the author remains the only Ph.D. student involved in this project and played the key role in the development of the methodology. Next, the contribution of the author is pointed out more specifically:

Separation of functionality and architectural decisions: The basic essence of the initial work on MASIC is the separation between the system functionality and the implementation architecture, and the creation of the grammar based language to build system models [*paper-vii, ix, xi*]. The former was mostly conceived by Dr. A. Hemani and Dr. J. Öberg, while the latter was mostly developed by the author of this thesis. The author is the only student involved in those papers, and he conducts most of the experimental works.

Modeling mixed hardware/software system modeling: The MASIC project—as the name suggests—was targeted to ASIC implementation of signal processing systems. To model a mixed hardware/software system, the methodology was extended to be able to deal with embedded cores [*paper-v*]. In MASIC, an embedded core is viewed at two different layers. The outer layer contains the *bus functional model* (BFM) of the core, and the inner layer contains the DSP function in C language. The author rendered the main contribution to develop the mixed hardware/software system design methodology and conducted the experiments.

Categorization of architectural decisions: The concept of the categorization of architectural decisions, which helps the designer to consider only a smaller subset of decisions at the beginning, was exclusively envisioned by the author [*paper-ii*]. The systematic categorization of architectural decisions made it possible to create intermediate abstraction levels in the design flow, which helps to tackle design challenges in small incremental steps. Prof. A. Jantsch played an important part in introducing and explaining to the author the role of different models of computation for modeling concurrency and time in a system model. The author developed the abstract models of the system, the detailed model of the AMBA on-chip bus architecture, and the BFM of the MIPS core to conduct the experiments presented in [*paper-ii*].

Mapping the functionality on a system architecture through a number of levels of abstraction: After the systematic categorization of the architectural decisions, the next obvious issue was to capture these decisions in an abstract way, and still being able to obtain as accurate performance estimates as possible. The concept of capturing the architectural decisions using abstract transactions through a number of transaction level models was solely developed by the author of this thesis [*paper-i*]. Prof. A. Jantsch and Dr. J. Öberg acted as mentors throughout the course of the work. The author conducted all the experiments presented in [*paper-i*].

MASIC Language: The creation of the *ProGram*[5] language [140] initiated the grammar based language development activity in the then *Electronic System Design* lab (ESD lab) at KTH. ProGram is targeted towards data communication protocols and control dominated systems. The MASIC compiler was built upon the principles of the ProGram compiler. The MASIC language is tailored towards high level modeling of signal processing systems. Except a couple of syntactical details, the major improvement of the MASIC over the ProGram is the ability to link to the C functions from the grammar description, such that a DSP function written in C can be fired to perform the computation. The author, with valuable feedback from Dr. J. Öberg, has developed the syntax and the semantics of the MASIC language, and the compiler to generate a high level simulation model in VHDL [*paper-viii*].

Petri net based simulation approach: This work was presented in [*paper-iii, iv*]. The author was the prime contributor of the work, and he received valuable feedback from Dr. J. Öberg and Prof. A. Jantsch. This work has made it possible to simulate the grammar descriptions without having to compile them to another language.

System verification through divide and conquer approach: Verifying individual DSP blocks together with the global control results in a complex validation problem. Taking advantage of the clean separation between the functionality and the architectural decisions of the MASIC model, this verification technique is developed to ease the verification task. This work was presented in [*paper-vi, x*], where Dr. A. Hemani and Dr. A. Postula made the prime contribution by formulating the idea. The author applied the idea in the context of the MASIC methodology.

[5] ProGram stands for *Protocol Grammar*, which was developed by Dr. J. Öberg

1.6 Organization of the Thesis

This thesis is organized as follows:

In the second chapter we discusses some of the pioneering works in the area of system design, which is needed to position the current work in the right context. To study the huge volume of works conducted in this area, we categorize the works in three broad areas: modeling, design and analysis, and compare/contrast our approach with the others.

Next, in chapter three, we present the MASIC methodology in details. The three cornerstones of the methodology are the clean separation of the functionality from the architectural decisions, the systematic categorization of architectural decisions, and the stepwise capturing of the architectural decisions through a number of abstraction levels. In this chapter, we elaborately discuss all these aspects of the methodology and point out the benefits of our approach.

Chapter four applies the MASIC modeling technique on several examples. We also present the grammar based language of MASIC that helps to capture and inject the architectural decisions efficiently. The modeling concepts are illustrated by realistic examples, which are implemented using the AMBA on-chip bus architecture. Our results show that the abstract TLMs can be built and simulated much faster than the implementation model at the expense of a reasonable amount of simulation accuracy.

Chapter five presents the system validation approach with MASIC. First, we discuss the divide and conquer approach to system verification, and discuss its benefits. Next, we present the Petri net based simulation technique of the MASIC description of a system model.

Finally, in chapter six, we conclude the current work and point out the direction towards future improvements.

2. SYSTEM DESIGN FOR SIGNAL PROCESSING APPLICATIONS

This chapter discusses some of the key activities conducted in the area of system design methodology, briefly tells how different researchers have tackled different issues, and along the way compares and contrasts the MASIC methodology.

2.1 Introduction

This chapter provides a review of some of the pioneering works in the area of system design methodology. Design methodology refers to the sequence of steps by which a design process will reliably produce a design "as close as possible" to the design target while maintaining feasibility with respect to constraints [3]. Methodologies create the need of different design tools, and the success of a design methodology largely depends on adequate tool support. Design tools are developed to perform the design steps with ease. The tools are built on the basis of different computational models, scheduling techniques, etc. These underlying models do not allow us to build any arbitrary type of tools. Hence, at times we also see that methodologies are being developed taking the available tool support into account.

To study the huge volume of works conducted in the area of system design methodology it is necessary to put the works in the right context of the design activities, rather than discussing them in isolation. It is, however, quite difficult to categorize these works because lot of the activities are tightly interwoven. Nonetheless, we classify them into three broad categories: *modeling*, *design*, and *analysis*, and briefly discuss how different researchers have put emphasis on different aspects.

The task of *modeling* is to describe complex systems in a simpler way. A model is a said to be the simplification of another entity, which can be a physical thing or another model [76]. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. It is customary to build a number of models of a complex system at different levels of abstraction. At higher abstraction level, systems are typically modeled as a set of concurrent processes interacting with each other in an abstract way. The lower the level of abstraction is, the more detail the description is. The semantics of modeling tells us how different processes of a model cooperate with each other, how they communicate, and how they compute.

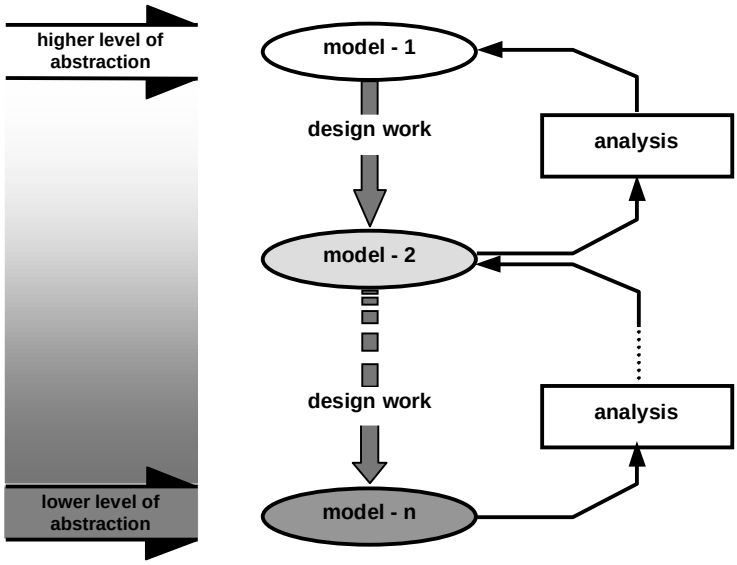

Figure 8. Different activities within design methodologies

It is the act of *design* that adds more details into a model and brings it to a lower level of abstraction. Figure 8 shows the conceptual framework of design methodologies where the design works successively bring a model from a higher abstraction level to a lower abstraction level.

Different models at different levels of abstractions are *analyzed* to validate their functionality, and to check their performance against the constraints. If the model does not meet its requirements then the modeling and the design work need to be performed again. Figure 8 shows the design analysis task using the arrows on the right side of the figure.

The design methodology tells us how all these different steps can be performed with ease so that an abstract model can be brought down to a detailed model.

2.2 Modeling

2.2.1 System Specification

A typical design methodology starts with the system specification model, and describes a design flow that leads to an implementation, as shown in Figure 9. Quite naturally, what is an implementation at a higher level of abstraction, is too far away for being termed as implementation at a lower level of abstraction. For example, a digital filter developed in MATLAB is an implementation for an application engineer but it is not an implementation for a system designer.

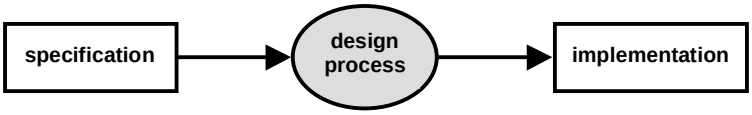

Figure 9. Design process

The specification model of a system typically contains two major parts [132]:

- A *functional specification*, which captures the input-output relation of a system without any implementation detail.
- A *set of constraints*, which contains non-functional information like, performance, power, size, design and manufacturing cost, etc.

It is unanimously accepted that the functionality of a system should be captured at the highest level of abstraction, without considering hardware/software partitioning or any implementation details. Thus the functional model relieves the designer to think about all the implementation details at the beginning, and gives the necessary freedom to explore the design space with different design decision, probably through a number of intermediate levels of abstraction, as shown in Figure 4, in chapter 1.

Different domain specific languages or tools with different underlying models of computation are used to specify the system functionality. We shall discuss more about models of computation in section 2.2.3. The Metropolis specification may mix declarative and executable constructs of their metamodel [14]. They use a logic language to capture the non-functional and declarative constraints. The specification model in the ForSyDe methodology employs a synchronous model of computation. The synchronous models are implemented using the concept of *process constructors*. They use the functional language *Haskell* to describe their models [117].

In general, system level languages are well suited for functional specification, but they typically lack in formalisms to incorporate the timing constraints. Generally the constraints are specified as a set of inequalities [50]. SpecSyn permits minimum and maximum constraints to be specified on behavior execution times and channel bit rates [54]. The SpecC methodology uses the SpecC language, which is built based on the ANSI C language [48]. Their specification model contains both the functionality and the performance constraints, where the constraints are captured in the form of annotations to the model [110].

There are advantages of decoupling the constraints from the functionality. For instance, the knowledge of functionality is not necessarily needed for scheduling and performance analysis purposes. It is enough to know the properties related to the resource requirement and interaction. This is the reason why the SPI (System Property Interval) workbench developed at the Technical University of Braunschweig uses domain specific languages to describe the system functionality, and annotates parameters to the model to capture the external behavior [77][139]. These parameters include data rates denoting the number of tokens consumed or produced by a process per execution on a certain channel, latency times denoting the time between the start and completion of a process, etc.

2.2.2 Concurrency and Communication in Dataflow

To express the concurrency and communication in signal processing systems, it is popular in the DSP community to model the system functionality using block diagrams, signal flow graphs, or data flow graphs. A block diagram consists of functional blocks connected with directed edges. The edges may or may not have delays. Block diagrams are a popular way of graphically representing DSP algorithms. For example, the 3-tap FIR filter described in equation-2.1 can be shown by the block diagram of Figure 10.

$$y(n) = c_o x(n) + c_1 x(n-1) + c_2 x(n-2) \qquad (2.1)$$

A signal flow graph (SFG) [33] is a collection of nodes and directed edges that models linear single-sample-rate systems. The nodes represent computations. A directed edge (*j*, *k*) denotes a branch originating from node *j* and terminating at node *k*. The edges are generally restricted to constant gain multipliers (the output of the edge is a constant multiple of the input signal), or a delay element (the output of the edge is a delayed version of the input signal). Adders and multipliers can be described by nodes with multiple incoming edges and one outgoing edge.

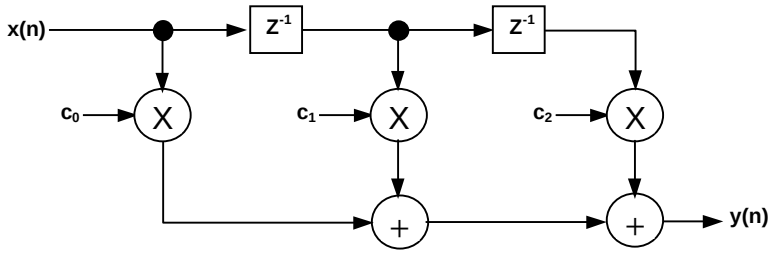

Figure 10. Block diagram of a 3-tap FIR filter

There are two special nodes in a SFG: a *source node*, which does not have any entering edge, and a *sink node*, which only has entering edges. Figure 11(a) shows the SFG of the FIR filter described by equation-2.1. Linear SFGs can be transformed into different forms without changing the functionality. Flow graph reversal or transposition is one of these transformations that is applicable to single-input-single-output systems. Transposition of an SFG is done by reversing the directions of all the edges, exchanging the input and output nodes, while the edge delay and gain remain unchanged. Figure 11(b) shows the transposed FIR filter, which is also referred to as data-broadcast structure. SFGs are extensively used in digital filter structure design and analysis of finite word length effects. However, SFGs are applicable to linear networks, they cannot express multirate DSP systems in a natural way.

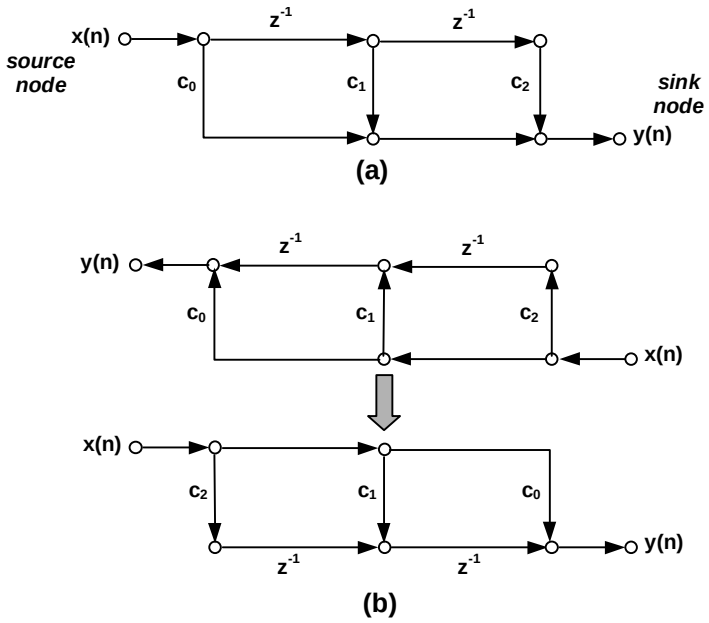

Figure 11. (a) SFG of a direct-form 3-tap FIR filter, (b) SFG of transposed FIR filter

Data flow graphs (DFGs) can be used to describe multirate DSP systems. They are widely used in high-level synthesis to derive concurrent hardware implementations of DSP applications. In DFGs, the nodes represent computations, and the directed edges represent the flow of data. Edges may or may not have delays associated with them. The nodes in a DFG can represent indivisible atomic operations or large grain operations. Figure 12(a) shows the DFG of the FIR filter described by equation-2.1. The empty circles are called *fork* nodes, which just replicate each input sample on all the output paths. Figure 12(b) shows the DFG of the transposed FIR structure.

The DFGs capture the *data driven* property of signal processing systems, that is the nodes can *fire* (i.e., perform its computation) whenever data are available on all the incoming edges. This implies that many nodes can fire simultaneously, which shows the concurrency in the system. Edges describe communication and precedence constraint between two nodes. If the edge has zero delay then this precedence constraints is said to be *intra-iteration precedence constraint*, and if the edge has one or more delay then this is called an *inter-iteration precedence constraints* [108]. The concurrency and communication, which exists in signal processing systems can conveniently be studied using the concept of models of computation.

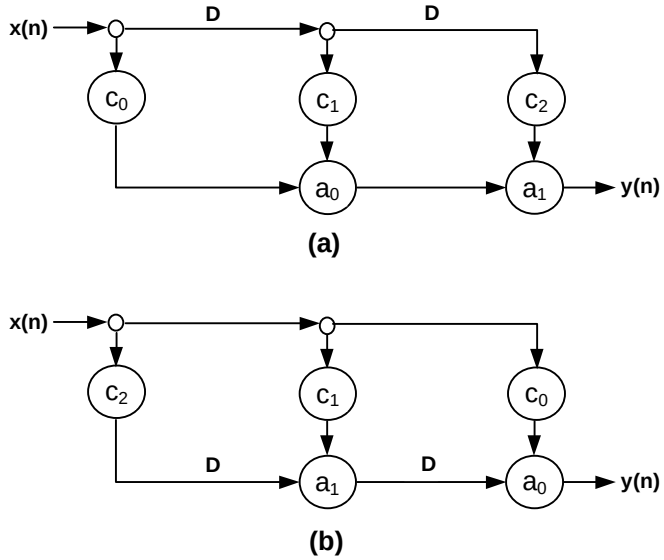

Figure 12. (a) DFG of a direct-form 3-tap FIR filter, (b) DFG of transposed FIR filter

2.2.3 Models of Computation

Concurrent systems consist of relatively autonomous modules that interact through messaging of some sort. In [57], Lee *et al*. have referred to the rules of interaction of these modules, and the semantics of composition as *models of computation*. Different models of computation are popularly used as the basis for designing signal processing software [93]. Here we shall restrict ourselves only to those models of computations that are popular for modeling signal processing applications. A detailed treatment of different models of computations can be found in [94].

A. Kahn Process Network

In 1974, G. Kahn wrote a small paper on the semantics of concurrent and communicating processes [78]. The model of computation that he described became known as the Kahn process network (KPN). Processes in a KPN map one or more input streams to one or more output streams, and they are connected through point-to-point (i.e., single reader and single writer) queues, which obey first-in first-out (FIFO) discipline. These FIFO channels provide the storage for the data, and the only means for the processes to communicate. Typically a

process goes through a non-terminating iteration, where each iteration cycle consists of reading token (i.e., data) from the input FIFO, performing some computation, and writing the result to the output FIFO. A process read from the input FIFO when data is available. Processes attempting to read from an empty FIFO will be stalled (i.e., blocking read) and they are not allowed to test an input FIFO for the presence of tokens without consuming the tokens. A blocked process may resume execution as soon as sufficient new tokens become available. Processes perform computation in their private memory and manipulate their own state space. The writing to the FIFO always succeeds, without ever getting blocked. This non-blocking write property in turn means that FIFOs can be theoretically unbounded. Graphically, processes are drawn as nodes and FIFOs as arcs. Figure 13 shows an example KPN, where unconnected arcs show input from and output to the environment.

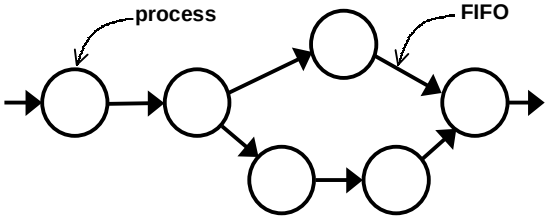

Figure 13. A Kahn process network

Task-level parallelism is explicit in a KPN in the sense that a process can execute independently when it has data in its input FIFO. It is, however, necessary to find out an order of execution of the concurrent processes of a KPN, for example to build a simulator of the network or to implement it on hardware. There are different techniques to find an order of execution of these concurrent processes. A highly cherished property of KPN is that the ordering of the execution of processes, that is the schedule, has no effect on the network behavior in the sense that given some tokens at the input FIFO the network will always produce the same output irrespective of the schedule.

B. Dataflow Process Network

Dataflow process networks are a special case of KPN [93]. In a dataflow process network, each process consists of repeated firings of a dataflow actor. The actors define the smallest quantum of computation. An actor can have a set of firing rules. Each firing rule consists of a set of patterns, one pattern for each of the inputs of the actor. An actor can fire if and only if one or more of the firing rules

are satisfied. The dataflow actors are *functional*, meaning that actor firing does not have side effects and the output tokens. Instead of stalling a process on a blocking read, dataflow processes can be freely interleaved by a scheduler, which determines the sequence of actor firings. Since the actors are functional, no state needs to be stored when one actor terminates and another fires. As a result, the context switching overhead due to process suspension and resumption associated to most implementations of KPNs is avoided [93].

C. Synchronous Data Flow

The synchronous data flow (SDF) model puts further restrictions on the behavior of the dataflow nodes in that the number of tokens consumed and produced during each invocation of a node is specified a priori [91][92]. Unlike the dataflow actors described above, the SDF nodes always have a single firing rule, and the rule does not specify the pattern of the tokens consumed. It only concerns about the number of the tokens consumed each time a node fires (i.e., performs its computation). In addition, an SDF node also produces a fixed number of tokens when it runs. The biggest advantage of this model is that the SDF nodes can be scheduled at compile time, rather that at run time, for a single or multi-processor implementation. As a result, the run-time overhead associated with scheduling is avoided. Multiple sample rates, which often exist in signal processing systems due to interpolation and decimation, can be easily handled using the SDF model of computation.

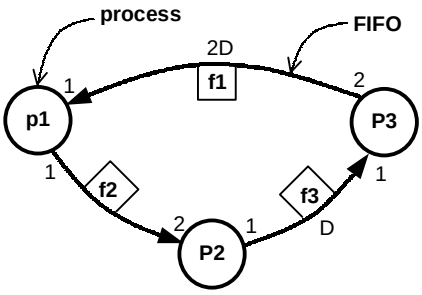

Figure 14. An SDF network

Figure 14 shows an example SDF network consisting of three synchronous nodes (or processes) called *p1, p2* and *p3*. They are connected to each other by three FIFOs called *f1*, *f2* and *f3*. Numbers associated with the processes specify the number of tokens (data) that they consumes from the input FIFO and produces to the output FIFO. For example, *p1* reads one token from *f1* and writes one token to

f2 when it fires. These numbers are part of the process definition. There may or may not be delays associated with the FIFOs. For example, *f1* has a delay of two and *f2* has no delay.

In this model, a unit delay on an arc from node *A* to node *B* means that the n^{th} sample consumed by node *B* is the $(n-1)^{th}$ sample produced by node *A*. This in turn means that if an arc with *m*-delay connects node *A* to node *B*, the first *m* samples consumed by *B* are not produced by *A* at all, but are part of the initial state of the arc buffer. In fact, a delay of *m* samples on an arc between two nodes of an SDF is implemented by putting *m* zero samples on that arc buffer at the beginning.

An SDF model does not include the connections between the network and the environment. For instance, the SDF network shown in Figure 14 only expresses the flow of data among in the processes of the network, but not the FIFOs to/from the environment. In this modeling style, a process with one input from the environment is considered as a process with no input, which can therefore be scheduled at any time.

D. Cyclo-static dataflow

Bilsen *et al*. extends the SDF model by introducing the cyclo-static dataflow (CSDF) where the number of tokens produced and consumed by a node is allowed to vary in a predefined cyclic way [20]. In CSDF every node has an execution sequence. Each time a node is fired it cyclically executes the corresponding sequence of execution, which in turn mean that the consumption and production of tokens are also sequences in a CSDF model. The GRAPE-II tool implements the static scheduling techniques for the CSDF model.

E. Dynamic dataflow and Boolean dataflow

Dynamic dataflow (DDF) actors are defined as the actors for which the number of tokens consumed or produced by a firing cannot be specified statically [26]. Typical examples of DDF actors are the *switch* and the *select*, which route tokens depending on a Boolean input.

Boolean dataflow (BDF) model lies between the SDF and the DDF model. Here, the number of tokens produced or consumed on each arc is either a constant or a function of a Boolean-valued token on an arc, called a control arc. The general problem of determining whether a BDF graph can be scheduled with bounded memory is undecidable [26].

2.2.4 Heterogeneous Modeling Style

The functionality of signal processing systems is data oriented where control is reduced to the extent of absence or presence of data in FIFOs. The process networks, described in the previous section, serve as the excellent basis for building the functional models of DSP applications. When we attempt to implement the functionality, we typically need to opt for a complex system architecture to meet the conflicting needs stemming from performance, programmability, power, and flexibility requirements. The system architecture often includes a heterogeneous mix of hardware software components connected through a shared communication medium, e.g., a bus. A bus interface is control oriented where data handling is rather trivial an operation, for example copying or emptying buffers. The dataflow networks are, however, awkward for specifying the control logic. Hence, we are faced with such a situation, where a modeling style needs to be devised that employs a heterogeneous mix of different models of computation, and languages.

Vissers *et al.* present the YAPI methodology, which targets system design for DSP applications. They employ a heterogeneous modeling style where the functionality is kept separated from the architectural decisions [44][84][131]. The MASIC approach is also similar. Other contemporary design methodologies, like the function architecture codesign [122] and the communication based design [83] also take a similar approach. The design flow in these methodologies takes the general form shown in Figure 15. Due to its distinctive Y-shape, it is often referred to as the *Y-chart* based approach. In such a design flow, a system designer typically studies the functionality, takes the system specification, makes a few initial calculations, and proposes an architecture to implement the functionality. Then the system performance is evaluated, for instance by simulation [84][131], and architectural decisions are altered to meet performance.

However, the Y-chart shown in Figure 15 has a top-down iteration, which is needed to meet the required performance by changing architectural decisions. Such a top-down iteration takes a lot of time, which hinders the design productivity. To speedup the design process and gain design productivity, it would be beneficial to break this top-down iteration into smaller loops by introducing intermediate levels of abstraction in the design flow. The importance of levels of abstraction is discussed in [56], where the authors argue that the key to the success of a methodology depends on well defined abstraction levels and models. The number of levels and the properties of each level have to be defined such that designers and tools can optimize decisions and move between the levels efficiently. The MASIC methodology adopts these observations to create an intermediate level of abstraction in the Y-chart based design methodology [41].

The basic idea here is to systematically classify the architectural decisions in two categories: *system level decisions* and *implementation level decisions*. First the system level decisions are added to the system functionality to create an intermediate model. Next, the implementation level decisions are added to the intermediate model to build a cycle true model of the transactions of the implementation architecture. As a result the top-down iteration, which exists in the traditional Y-chart based approach is broken into two smaller loops.

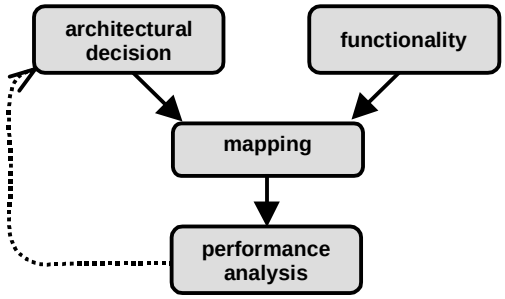

Figure 15. The generalized *Y-Chart* based approach to system design

The dichotomy of the communication and the computation parts of a system can be effectively handled by *transaction level models* (TLMs). Using different TLMs it is possible to ascertain what role communication and computation play at different levels of design abstraction. The concept of TLM first appeared in the domain of system modeling languages, like SystemC [59] and SpecC [48]. In [59], a TLM is defined as a model where communication between modules is modeled in a way that is accurate in terms of what is being communicated, but not in a way that is structurally accurate (i.e., the actual wires and pins are not modeled). They have also insisted that the communication between modules be modeled using function calls. Gajski *et al.* have defined a TLM as a model where the details of communication among components are separated from the details of computation within the components [28]. Communication is modeled by channels and transaction requests take place by calling interface functions of these channels. They also present the *system modeling graph* where computation and communication are shown in two axes. Each axis has three degrees of time accuracy: untimed, estimated, and cycle accurate. The MASIC methodology effectively employs the concept of transaction level modeling to design signal processing systems [42].

Heterogeneity exists in a system model due to various reasons. So far we have discussed the heterogeneity due to the functionality and the implementation

architecture of a system. Bolsens *et al*. argue that only the functional part of a system may call for a heterogeneous modeling style [23][114]. For example, a dataflow functional block (DFFB) operating on unfragmented signal words can be best specified as dataflow networks. When a DFFB manipulates individual bits of the signal (e.g., Viterbi channel encoding), then it can be conveniently specified as a finite state machine with datapath (FSMD). Hence there is a need of a heterogeneous environment that allows describing the functionality of different parts of the system using the language and models of computation that is most suitable to model that specific part. The simulation and synthesis of heterogeneously specified digital communication systems using the CoWare design environment, developed at IMEC, is presented in [23].

Girault et al. describe the **chart* formalism that allows hierarchical description of different computational models, like finite state machine, dataflow, discrete event, etc. Different dataflow networks, like synchronous dataflow, Boolean dataflow, cyclostatic dataflow can be represented in the *FunState* internal design representation [126]. They employ the internal representation for scheduling and performance analysis purposes. The Ptolemy environment, developed by Lee *et al*. at UC Berkeley, is specifically targeted towards simulation and prototyping of heterogeneous signal processing systems [27]. We shall discuss more on Ptolemy in section 2.4.1.

It is necessary to point out that, a heterogeneous modeling environment makes it easier for the designer to build system models using a suitable mix of different models of computation. However, scheduling a heterogeneous description of functionality becomes more challenging.

2.2.5 Languages

Languages are used to describe system models. There exits a huge variety of languages. Some languages are for building simulation models at a particular level of abstraction. Some languages build synthesizable models, that is they allow the model to move from a higher abstraction level to a lower abstraction. Often we find languages, for example VHDL, where only a subset of the available language constructs is synthesizable, while the other constructs are not.

The description style of a model can be categorized into two categories: *homogeneous* and *heterogeneous*. A homogeneous description style models a system using a single language, which implies that the language needs to be expressive enough to model different aspects of a system. Languages are built

upon different models of computation. As a result, often we see that the semantics of a single language is not suitable to efficiently express different aspects of a system. For example, a dataflow flow language can very easily describe a signal processing functionality but it lacks in expressiveness to describe the system control. A heterogeneous description style models a system using more than a single language. Here the prime advantage is that the designer can use different suitable language to express different aspects of a system. However, the problem with a heterogeneously specified model is to coordinate the computation and communication among different modules described in different languages.

The Silage language is developed for specifying complex signal processing applications. Silage nice expresses dataflow description, and supports a few conditional constructs to describe iteration and condition [55]. This work has been commercialized as the DFL language in the Mentor Graphics DSP Station environment. The synchronous languages, like Signal [60] and Lustre [66], are developed to describe data intensive applications. The word synchronous is often used in the literatures to refer to at least three different concepts, depending on the context of discussion [31]. In the context of models of computation, synchronous refers to a system where the components react instantaneously to an event, that is the reaction time is zero. In the context of communication protocols, it refers to a protocol without an acknowledgement, that is the sender does not wait for the receiver to acknowledge for the reception of the message. In the context of hardware, it means a circuit where the components are synchronized by a global clock signal.

To augment the expressiveness available within a single language researchers often extend it with an existing language with library routines. For example, the Scenic design environment provides a C++ class library, which enables a designer to express the reactivity of a hardware system [97]. The Scenic environment has evolved into the SystemC language. SystemC is also built upon C++ where they supply a class library to be able to model concurrency and time [59]. The SpecC language is based on ANSI-C and they provide special extensions (keywords) to model a system [48]. Both SystemC and SpecC provide the concept of interfaces and channels, which are an abstraction of signal level communication. As a result, systems can be modeled in a way such that the computation and the communication do not get mixed up. If a component of a system is designed in a way such that the computation and the communication get mixed up then it becomes difficult to reuse the component because the functionality of the component depends on the communication pattern of the component. However, there are differences between these two languages. For example, the SystemC language provides a rich modeling environment where it is possible to add user-defined classes to model certain aspect of a design. While such flexibility makes

simulation easier, it is not possible for the CAD tools to support different concepts represented in different constructs. As a result, the support of CAD tools is limited to only a subset of the language. The SpecC language cannot be extended by the user. As a result, they can avoid the situation where CAD tool support is restricted only to a subset of the language.

There are quite a few system design languages based on different object oriented languages. For example, Rosenstiel et al. has described a Java based system simulation environment [86]. The OCAPI design environment uses the C++ language to model systems, and includes a set of object definitions that allow describing hardware at RT level [129]. OCAPI descriptions can be automatically translated to VHDL.

Grötker et al. point out the importance of having a unified design environment for modeling mixed control data flow system and introduced a modeling technique based on *process coordination calculus* (PCC), which defines a coordination language [58]. It combines the notion of multirate dynamic dataflow graph with the event driven process activation. Thus it maintains the flexibility and expressiveness of dataflow representations, while enabling designers to efficiently control these operations by incorporating control automata.

The MASIC description of model includes a heterogeneous mix of C functions and grammar notations. The functionality of a signal processing system is computation intensive, and we use C functions to perform the computational part of the system. The implementation architecture mostly involves control and communication, and we use grammars for this part of the model. The communication among the C functions and the firing of the functions are described using grammars.

2.3 Design

2.3.1 System Partitioning

After an abstract model of a system is built, the next question that comes into mind is how the design process would bring this model to an implementation. A huge body of research has been conducted on system partitioning into hardware and/or software components. The reason for hardware/software partitioning stems from the fact that, while it is necessary to implement some part of the system

functionality in hardware to meet the performance constraints, like speed and power, it is preferable to implement some other parts in software to achieve programmability. Programmability offers flexibility, which is of crucial importance in designing modern signal processing systems, like mobile devices and set top boxes that need to support multiple standards.

In literature, the terms *system design* and *co-design* are often used interchangeably. Hardware/software co-design means meeting system level objectives by exploiting the synergism of hardware and software through their concurrent design [45]. The role of hardware/software partitioning is the optimized distribution of system functions to hardware and software components [53]. Along with partitioning of the system onto components, we need to perform scheduling to determine the execution order of processes on processors.

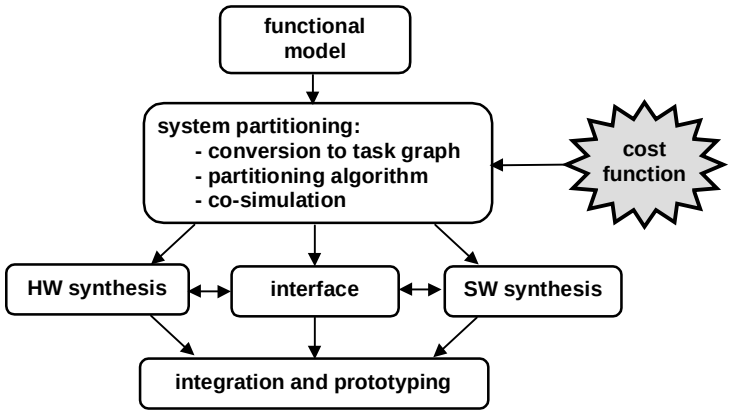

Figure 16. A general framework of a design flow based on partitioning

A general framework of a design flow based on different portioning approaches is shown in Figure 16. Typically a functional model is converted into an internal representation (e.g., a graph model) and an algorithm is used to implement different vertices of the graph in different resources to minimize a given cost function. The cost function can be a combination of time, area, power, size, etc. The importance of the cost function must not be under-estimated, and we will discuss more on it in section 2.3.2. The mapping of different nodes of the graph on different architectural resources is often viewed as a scheduling problem. We shall discuss a few scheduling techniques in section 2.3.3. Design methodologies provide a unified framework to perform hardware/software co-simulation often at different levels of abstraction. The main advantage of such a design flow is that it

allows concurrent development of the hardware and software components, and facilitates the interaction and feedback between the two. It is a major improvement over the conventional design flow where hardware and software development take place independently and without any interaction. In this case, the pitfalls become visible too late in the design flow, only after system assembly when the software is run on the hardware.

Vulcan is one of the early codesign methodologies, which uses the HardwareC language to capture the algorithmic description of the system functionality [61][62][63]. HardwareC has a C-like syntax and supports timing and resource constraints. The HardwareC description is converted into data-flow graphs. Their algorithm started with a solution of all operations in hardware, and then operations are selected to move to software to minimize cost. Such an algorithm strives to achieve maximal number of operations in software. In contrast to the hardware oriented approach of Vulcan, COSYMA [52] and AKKA [75] employ a software oriented approach where all operations are initially mapped to a CPU and then some parts are moved to hardware to meet timing constraints. The software oriented approach has the advantage that the model can easily contain complex data structures like dynamic data structures. COSYMA uses the C^x language, which extended the C language with the concepts of timing and task. The system description in C^x is translated into an Extended Syntax graph (ES-graph) from where different nodes are marked to be moved to hardware. They introduce a dual loop approach where a simulated annealing algorithm is used at the inner loop to determine partitioning decisions.

Jerraya *et al.* presents the COSMOS system where the system specification is done in the SDL[6] language. They translate this description to an internal representation called SOLAR, and produce a heterogeneous architecture including hardware description in VHDL and software description in C [74]. Eles *et al.* presents a simulated annealing and tabu search based technique where hardware/software partitioning is performed in four steps: extraction of blocks of statements, loops and subprograms, process graph generation, partitioning of the process graph, and finally, process merging [51]. Rosenstiel *et al.* presents a two stage clustering algorithm to partition a system specified in the UNITY language into hardware and software components [15].

Kalavade and Lee present a codesign methodology for DSP applications using the Ptolemy framework [80]. This paper describes the simulation, prototyping and software synthesis of signal processing systems. Hardware/software partitioning is done manually. In a subsequent paper they present an algorithm for the

[6] Specification and Description Language

constrained partitioning (assignment and scheduling) [81]. The application is described in a DAG where nodes represent computations and arcs describe data and control dependencies. They define the hardware/software partitioning problem as to find a mapping (assignment) of nodes to hardware and software, and the start time of each node (scheduling).

Since partitioning solutions affect scheduling results, an integrated approach to hardware/software partitioning and scheduling is presented in [101]. Madsen *et al.* argue that the optimal partitioning of functionality on processors cannot be done without considering the protocol of data transaction between the processors over the communication architecture [85]. They extended the LYCOS codesign methodology to incorporate the communication protocol selection as a parameter within hardware/software partitioning. In addition to performing *spatial* partitioning (i.e., dividing specification into hardware and software), Gajski *et al.* present a *temporal* partitioning approach in which the specification is divided into pipeline stages [13].

Vahid *et al.* points out the importance of the estimation of design cost due to hardware size [128]. The most accurate estimation could be found by synthesizing the design. Synthesizing hardware during design space exploration among hundreds of potential partitioning possibilities, however, would take computationally prohibitive amount of time. Hence they present a fast and reasonably accurate, hardware size estimator to help the designer to explore the design space. Henkel *et al.* also presents a high-level estimation technique for hardware/software partitioning [69]. They argue that considering a fixed granularity (either small system parts, or large system parts) during partitioning is likely to produce suboptimal solution. Therefore, they present a flexible partitioning approach.

In MASIC, system partitioning is done manually. We describe the partitioning and mapping information using design decisions. To facilitate the designer with different partitioning decisions, we capture the design decisions using abstract transaction. As a result, it becomes easier to change the partitioning decisions when performance requirements are not met.

In general, partitioning of the system functionality is a very complex task. There exist a variety of system components, and a number of architectures to interconnect them. This enormous variety of possibilities makes the search space too wide for a partitioning method. As a result, partitioning solutions often target a fixed architectural template, like a single CPU cooperating with a couple of hardware blocks [62][63][75]. Gajski *et al.* [54] and Wolf [133] present methodologies that target arbitrary combinations of CPUs and hardware blocks,

and arbitrary interconnection topologies. To this end, the main issue is that the tool needs to know how to partition functionality among the variety of components and how to estimate for such a partition.

2.3.2 Reuse Methodology

In the previous section, we have introduced the concept of a cost function and pointed out the importance of cost while partitioning a system. Partitioning splits the system functionality to achieve an optimal distribution of hardware/software components while minimizing the cost function. Typically, partitioning suggests for some blocks to be synthesized in hardware. However, when the design cost of a new hardware overweighs all the other constituents of the cost function then we need to put in place a design methodology that favors *design* reuse and *design for reuse*.

The reuse methodologies are increasingly gaining momentum due to the practical realities. Market research conducted by Synopsys Inc. has shown that the demand and potential profit for a new hardware/software product can be modeled by a triangular window of opportunity [43]. In order to maximize profit, the product must be on the market by the start of the demand window. As a result, the system designers face an extreme time-to-market pressure and want to have a methodology that would facilitate reuse of pre-designed and pre-verified IP blocks and software components. This desire is further aided by the fact that, in reality, a SOC design seldom starts from scratch. Along with the system functionality, the system designer in industry often finds a reference point for implementation in the form of a hardware solution from the previous generation, a legacy code, etc.

Though putting together pre-designed IP blocks were thought to be like putting together Legos, soon it turn out to be much more difficult because the IPs do not have a standardized interface as the Lego blocks do. To ease the reuse of IP blocks from multiple sources, industry consortiums like VSIA [8] and VCX [7] are created in order to define standards and rules for IP reuse and trading.

To tackle the design reuse and verification challenges, researchers have proposed the interface based design methodologies [114][116]. Typically a design team would design a system block in a way where the control logic needed to implement the signal processing algorithm is interwoven with the control logic needed for the data communication among different functional blocks. If a component is designed in such a way where the component behavior depends on the communication mechanism, it becomes difficult to achieve *design for reuse*.

On the other hand, a modeling style, which mixes the communication and the computation aspects of a system does not favor *design reuse*. The key idea of interface based design is the separation of the communication and the computation aspects of a design. Keutzer *et al.* have elaborately discussed the separation between these two design issues [83]. They have explained that function (computation) is an abstract view of the behavior of the system and characterized by its input-output relation. The implementation architecture (communication), on the other hand, states how the function is implemented. Separation between these two aspects splits the design task into smaller and more manageable problems, which in turn helps to explore the design space with alternative solutions. Their ideas are commercialized in the Cadence Virtual Component Codesign (VCC) tool.

The VSIA attempts to standardize the definition of interface based design at different levels of design abstraction. They have developed the System Level Interface Behavioral Documentation (SLIF), the On-Chip Bus Virtual Component Interface (OCB VCI), and the system level data type standard. These standards are used to design industrial products for telecommunication and multimedia applications [96].

Synopsys and Mentor Graphics signed the Design Reuse partnership agreement to support reuse methodology, which resulted in the publishing of the Reuse Methodology Manual for SOC Designs [82]. The classic top-down design methodology goes through a specification, decomposition, integration and verification, which resembles a *waterfall* model of system development. A top-down methodology assumes that the lowest level blocks can actually be designed and built economically. If it turns out that the block is not feasible to design, the whole process is repeated. To avoid such repetition and meet time-to-market pressure, many design teams are moving from the waterfall model to the *spiral development model*, which favors design reuse and facilitates designers to work simultaneously on different aspects of the design. The Set Top Box designed at STMicroelectronics is developed using a design flow similar to the spiral model [113].

The platform based design methodology [83][119] takes another step forward towards design reuse by providing a domain specific programmable solution to meet the aggressive performance requirements that a general purpose programmable solution is not likely to achieve. The MESCAL project intends to develop the methodologies and tools to realize platform based solutions for specific high volume application domains by providing programmable solutions that can be used across multiple applications in that domain [83][102]. A hardware platform is a common hardware “denominator” that could be shared across

multiple applications in a particular domain. A family of such hardware would allow substantial reuse of software. The hardware platform is wrapped by a software layer (RTOS, device drivers, etc.), which is known as the software platform. The combination of the hardware and the software platform is called the system platform. An important part of the architecture is the development of the communication network tailored to the application domain. The architecture exploits concurrency at all levels—bit level concurrency, instruction level parallelism, and thread/process level concurrency. The compiler development for this system is quite challenging, as it requires retargetable capability for compiling to different processing elements within the same family, as well as synthesizing the communication and synchronization code for thread/process level concurrency. A platform based design methodology, *derivative design*, has been used by the mobile phone industry [105]. Instead of introducing new mobile phone designs multiple times a year, developers introduce derivative designs into different market segments. Due to the intention of supporting multiple applications in the same domain a hardware platform may become over-designed for a given product, that is, some of the power of the micro architecture might not be used. The proponents of this methodology argue that, the over-designed hardware platform would soon be popular in embedded systems design community to deliver new software products and extend the application space.

MASIC effectively supports design reuse. The communication and the computation aspects are kept separated in the MASIC model. It allows to build system models using the interface description of predesigned and preverified hardware blocks.

2.3.3 Scheduling

The data driven networks discussed in section 2.2.3 provide an excellent way of modeling concurrent tasks. However, they cannot be executed as such on a CPU. Given the loosely coupled data driven networks, the essential question is how to write a simulator for this network, or how to map them onto computational resources to synthesize a system. Traditionally, the mapping of functions on the computational resources is viewed as a scheduling problem. Scheduling is an important issue regarding an optimum implementation of these networks that determines which process executes on which resource at which point in time. There exist different scheduling techniques that determine the order of execution of the concurrent processes of these networks. It is popular to classify scheduling algorithms in three categories: dynamic scheduling, static scheduling and quasi-static scheduling.

A. Dynamic Scheduling

In the general case of computing, a run-time supervisor determines when processes are ready to be executed, and schedules the processes onto the processors when they become free. Liu and Layland describe priority driven scheduling techniques for time-critical control and monitor processes [100]. If the priorities are statically assigned to processes once and for all such a technique is said to be *fixed priority scheduling*. They propose to assign priorities to processes according to their request rates, independent of their run-time. They assume that the processes are invoked periodically and assign higher priority to a process with higher request rate (i.e., shorter invocation period). Such a scheduling technique is known as *rate-monotonic scheduling* (RMS). To achieve a better processor utilization, they also propose the deadline driven scheduling technique where priorities of processes are not fixed. The priorities are assigned dynamically based on the deadlines of their current requests and hence this scheme is often called *earliest deadline first* (EDF). In this scheme, highest priority will be given to a process if the deadline of its current request is the nearest. Both RMS and EDF scheduling support *preemption*, that is the processing of any process can be interrupted by a request for another higher priority process. It is necessary to note that preemptive scheduling is not usually employed in embedded systems to avoid unpredictability and scheduling overhead, and to minimize cost. The cost penalty is due to the memory requirements to implement the stacks needed to store the states of the preempted processes.

Apart from the general techniques discussed above, several specialized scheduling techniques have been developed based on better knowledge of the domain. If we consider the KPN shown in Figure 13, the problem of scheduling it on a multiprocessor system is to determine which process executes on which processor at which point in time. A special case is a single processor system where all the processes are executed sequentially. The basic approach to scheduling a KPN is to choose as many *ready* processes for execution as there are free processors available. Two distinct scheduling approaches have been proposed to determine the set of ready processes at any point in time, namely *demand driven scheduling* and *data driven scheduling*.

In demand driven scheduling [79], processes are scheduled on the basis of demand of data that originates at the output of the network. A single *driver process* (one with no output) demands data as shown in Figure 17. When an input FIFO runs out of data, the process is blocked and the input FIFO is marked *hungry*. Now the demand propagates through the network. The process responsible to *put* data on the FIFO becomes ready process, which in turn may become blocked if its input is empty as well. The demand propagation between *p1* to *p2* is shown by the dotted

arrow in Figure 17, when no data is available in *f1*. If a process produces requested data to meet the demand of a hungry channel, it will be suspended and the destination process will be restarted where it left off. Due to the propagation of demand, processes often become blocked. The processor executing a blocked process needs to make a context switch to start executing another ready process. This style of scheduling is, therefore, known to suffer from expensive context switching overhead especially when relatively little computation is done each time a process is run.

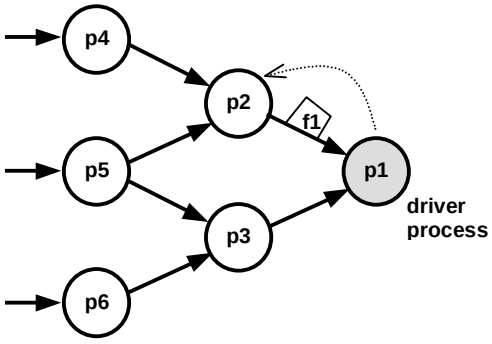

Figure 17. Demand driven execution of a KPN

The data driven scheduling approach, on the other hand, schedules a process on the basis of the availability of data. Initially, data is supposed to be available at the input of a network and the processes consuming these data are the ready processes. A process runs as long as it has input data available. Execution of the ready processes produces new data, which in turn makes other processes ready. Though this style of scheduling avoids context switching, it may lead to unbounded execution. Since FIFOs are theoretically unbounded, input processes can continue running as long as data are available without scheduling other processes using the data produced by the input processes. As a result, the system will run out of memory without ever producing an output. A way around this problem is to bound the sizes of the FIFOs and to perform a data driven process execution as long as there are space available in the bounded FIFOs. To this end, the problem is to find the bounds of the FIFOs. If the bound is chosen too large then the system occupies more memory than needed, and if the bound is too small then the system performance would deteriorate due to frequent context switches. In addition, *artificial deadlocks*[7] could be introduced in the system that might not have been

[7] The execution of a process network is said to *deadlock* when all the processes block on full or empty FIFOs. If at least one process blocks on a full FIFO, it is said to be an artificial deadlock [16].

taken place if the FIFOs were larger. Basten and Hoogerbrugge propose an algorithm to find the sizes of the FIFOs at design-time [16]. They start with a small FIFO size and go through a number of simulation runs with different input data to determine the necessary FIFO size so that an artificial deadlock can be avoided. However, the FIFO sizes determined in this way cannot guarantee a deadlock free execution because a different set of input data than the ones used in the simulation may lead to an artificial deadlock.

B. Static Scheduling

When all the scheduling decisions are taken at compile-time or design-time then the system is said to be statically scheduled. In general, a static schedule cannot be found for dataflow networks. For SDF networks, however, Lee *et al*. have developed efficient techniques for constructing both a sequential schedule (for single processor implementation) and a parallel schedule (for multiprocessor processor implementation) [91]. They construct the *topology matrix*, which is similar to the *incidence matrix* of directed graphs, of an SDF network. Using the rank of this topology matrix, they show the necessary condition for the existence of a sequential schedule, and describe an algorithm to build such a schedule, if one exists. To build a parallel schedule, they build an acyclic precedence graph and employ a heuristic method to schedule the nodes of the SDF graph. The FIFO sizes between the nodes of an SDF graph can be determined at compile time. Both the sequential and the parallel schedules are periodic in that they produce and consume the same number of tokens on each arc through an iteration of the schedule. As a result they can be repetitively applied on an infinite stream of data with bounded FIFO.

The SDF scheduling technique, however, has few limitations. For example, it does not consider connections of the SDF graph to the environment. As a result, buffering of input data is assumed to be a part of the environment. It is, however, often desirable to schedule a process to collect inputs so that the buffering requirements at the input can be minimized. Again, the run-time of a process is assumed to be known a priori. Though the run-time does not play any role in the sequential schedule, it is needed for the construction of a parallel schedule. If a process has a data dependent behavior then it is clearly not possible to know the run-time of the process in advance. However, even for the data dependent process execution, the computation time is bounded in hard real time applications.

Several commercial companies offer software packages for modeling signal processing systems based on the SDF model. Examples are the COSSAP tool from Synopsys, and the Signal Processing Workstation (SPW) tool from Cadence. The

key advantage here is that operations can be scheduled statically at compile-time, and as a result, a model can be simulated orders of magnitude faster than the event driven simulation of a (V)HDL model.

It is also possible to construct a static schedule of the CSDF network. In [20], Bilsen *et al*. have presented the static scheduling technique of CSDF networks for both single processor and multiprocessor systems.

C. Quasi-static scheduling

Quasi-static scheduling is in between the dynamic and static scheduling approaches, where some of the scheduling decisions are made at run-time whereas the others are done at compile-time. Ha and Lee present a quasi-static scheduling technique of dataflow networks with dynamic constructs [65]. They argue that the static scheduling does not have the ability to balance the computational load of the processors running processes with conditional and data-dependent iteration. Though the dynamic scheduling has the ability to handle such issues, the run-time overhead may be excessive. To minimize this overhead a few decisions are made at compile-time based on the available known profiles of dataflow actors. The profile of an actor is the static information necessary for scheduling, for example the execution time and the communication pattern. For the dynamic constructs they define a cost function, try to minimize the cost function, and find a near-optimal schedule.

2.3.4 Communication Refinement and Interface Adaptation

With increasing system complexity and time-to-market pressure, it becomes necessary to design system at the processor-memory level, not at the RT level. At higher level of abstraction, communication between processes takes place through abstract channels and abstract interfaces, as shown in Figure 18. As the design flow moves towards implementation, the computation part of the system is mapped onto system components and the communication part is mapped onto memories and interconnection architectures. Communication refinement addresses the issue of mapping the abstract communication channels on the actual wires and pins of an architecture. The major problem of reusing off-the-shelf components is the variety of communication protocol that they use. When components with incompatible protocols need to be interfaced, protocol conversion is needed. Interface adaptation deals with interfacing a component with the actual wires and pins of a communication architecture.

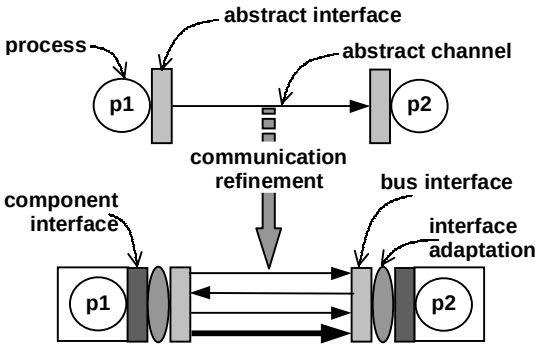

Figure 18. Communication refinement and interface synthesis

DSP applications are often modeled using Kahn Process Networks (KPN) [78], or different forms of dataflow networks, like SDF [91][92], DDF [26], etc. Processes in a KPN are connected through infinite length point-to-point FIFOs. The communication refinement of a KPN using the COSY communication IP's is presented in [24][25]. They define two types of transactions: *application level transaction*, and *system level transactions*. The former is used to program a network of functions that capture the functionality of the system and the latter is used to capture the architectural model of the system. At application level, communication takes place through FIFOs. At system level communication takes place through buses. A trace based communication refinement technique is presented in [98]. They also use KPN to model a signal processing application. Their trace transformation technique provide the mapping between application level communication primitives and architectural level communication primitives.

Gajski *et al.* presents communication techniques for system level design [9][110]. There are three inputs to their communication refinement engine: the concurrently executing system components exchanging data via abstract channels, the protocol library that supports a variety of protocols including generic and processor specific protocols, and the set of user given synthesis decisions, like bus allocation, connectivity, bus priority, etc.

Jerraya *et al.* presents a component based design methodology in which they present a communication refinement approach [30]. They build an abstract virtual architecture where virtual components communicate through FIFO channels. Their virtual components represent both a hardware function or software task. They provide an automatic wrapper generation tool, which synthesizes hardware interfaces, device drivers and operating systems to implement the system.

The high-level compiler called *Integral* developed at IMEC synthesizes interface modules with automatic protocol conversion [99]. The input to their tool is a high-level concurrent algorithmic specification that can be represented as an annotated Petri net model extended with CSP-style[8] rendezvous communication. Their compiler accesses a library containing the actual I/O protocols and timing information of different components. The I/O protocols are specified using *signal transition graphs*, which is also a Petri net model but with only signal transitions as possible actions. Given the algorithmic specification of the system and the protocol library, the necessary protocol conversions are automatically synthesized using a Petri net theoretic approach. The Integral tool is incorporated into the CoWare design environment, which is used to design numerous signal processing systems [23][114].

Passerone *et al.* also address the issue of interfacing IP's involving incompatible communication protocols by obtaining an FSM that when placed between the two modules would make the communication possible [109]. They start from a regular expression based protocol description. The regular expressions, representing the two protocols, are translated into two automata that recognize the corresponding regular language. Finally, the product of the two automata is taken to capture the legal sequence of operations.

2.4 Analysis

2.4.1 Simulation

Simulation remains the most widely used means of system verification. The simulation of homogeneous model is a relatively simpler task. System simulation becomes more difficult for heterogeneous models. In this context, two issues are of major concern: how to synchronize the simulation when the system contains a heterogeneous mix of hardware and software components, and how to synchronize the simulation when it contains a heterogeneous mix of descriptions based on different models of computations.

System simulation with hardware and software components is referred to as cosimulation. Hardware/software cosimulation verifies that hardware and software components function correctly together. There are different ways of hardware

[8] CSP stands for *Communicating Sequential Process* [73].

software cosimulation, depending on various tradeoffs between a number of factors, including [115]:

- Simulation performance
- Timing accuracy
- Model availability
- Visibility of internal state for debugging purposes

If we want to increase timing accuracy of the simulation then we have to simulate the system at lower level of abstraction, which is very slow. Fastest simulation can be achieved using emulators, which runs only slightly slower than the real hardware. However, in this case the turn-around time can be longer. Model availability also plays a role. For example, the *bus functional models* (BFMs) of processor cores are used when the actual core is not available. The core BFMs represent the structurally accurate interfaces, and they perform the cycle accurate transactions (e.g., read, write, burst read, burst write, etc.) that the actual cores make. As a result, the BFMs are the most widely used technique to debug hardware [115]. However, the BFMs do not include the internal architecture of the processor datapath, and software simulation at instruction set level is possible with them. In the experiments presented in the dissertation we also use BFM of MIPS processor core to perform system simulation.

System simulation of heterogeneously described models is elaborately addressed in the Ptolemy environment, developed by Lee *et al.* at UC Berkeley [27]. Ptolemy is specifically targeted towards simulation and prototyping of signal processing systems. For the purpose of synthesis, it can synthesize assembly code for programmable DSP cores. Ptolemy is a coordination framework that is not restricted to any particular computational model, instead it coordinates synchronous dataflow (SDF) networks [91][92], dynamic dataflow (DDF) [26], Boolean dataflow (BDF), discrete event, communicating sequential process (CSP) [73], etc. They use the object oriented class definition to model each subsystem. The basic unit of modularity in Ptolemy is a block. Blocks communicate through interfaces provided by portholes. The lowest level objects in Ptolemy are of the type star, derived from blocks. A galaxy, also derived from block, contains galaxies and stars. A universe, which contains a complete Ptolemy application, is of type galaxy.

Bjuréus et al. present a synchronization technique to simulate heterogeneous systems containing control and dataflow [21]. In their modeling environment, the

data intensive part of the system is described in MATLAB whereas the control intensive part is described in SDL. They use the composite signal flow model of computation to perform a cosimulation between MATLAB and SDL.

2.4.2 Performance Estimation

Based on how a system interacts with its environment, it is popular to divide them in two categories: *interactive systems* and *reactive systems* [18]. In interactive systems, the pace of interaction between the system and the client is determined by the system. In this case, clients wait to be served. In reactive systems, on the contrary, the pace of interaction is determined by the environment not by the system. Large systems, of course, can have components of both kinds. For example, maneuvering an airplane is mostly reactive (the roll, pitch and yaw of the plane must be adjusted according to the need of the environment), while communication with the ground control is mostly interactive. Embedded systems often fall under the category of reactive systems where the main concerns are timeliness and correctness. In one hand, if a system fails to respond within the required time then it is of no use. On the other hand, if it responds earlier than the required time, that does not add any extra value to the system. Such strict requirement of timing constraints on embedded systems makes it clearly different from a general purpose design, where the increase in every single GHz adds extra value. Consequently, performance estimation is considered as a key concern of system design. Here we use the term performance to refer to the temporal behavior of the system. For streaming applications, for example multimedia, performance is often measured in terms of latency and throughput.

To estimate the performance, a model is analyzed to assess one or more attributes of a system. The role of a design methodology is to provide a path to the designer so that a system model can be improved until it reaches the desired performance goals. Researchers have developed performance estimation techniques and tools, which estimate performance of a model at different levels of abstraction and often trade estimation accuracy against execution speed.

Estimation techniques can be divided into three categories [22]: static, dynamic and hybrid. A static estimator does not require an execution of the model. Often it relies on statistical and probabilistic techniques. A static approach often provides conservative estimations because hidden dependencies and false paths are hard to find, and the analysis needs to cover all alternatives in a given search space. This technique is good for worst case analysis. A dynamic estimator relies on the simulation of the system model. There are two ways of dynamic estimation:

run-time and off-line. The run-time estimator collects data and estimates performance during simulation time. For off-line estimation, the model is first annotated with estimation data, which is then used during simulation. The dynamic estimators are less conservative in that the hidden dependencies and false paths will never ruin the estimation. However, finding the corner cases is difficult with this approach, unless an exhaustive simulation is performed. Hybrid estimators borrow a bit from both static and dynamic approaches. For example, the trace based hybrid estimation techniques presented in [88] require that the model is simulated, and during simulation a trace is generated. Later the estimator analyzes the simulation trace to estimate performance of on-chip communication architecture.

2.4.3 Power Estimation

Low power/energy consumption is of crucial importance for both handheld and other systems. Therefore, the power and energy constraints must be tackled at every levels of abstraction.

Lajolo et al. present power estimation techniques for hardware/software systems design [90]. Their estimation is based on system level cosimulation. In [68], minimization of power dissipation has been addresses through hardware/software partitioning. Power analysis and minimization techniques for embedded DSP software have been presented in [95]. A comprehensive survey on system level power management techniques can be found in [17], and an overview of power conscious CAD tools and methodologies can be found in [125].

2.5 Conclusion

This chapter has discussed some of the pioneering works in the area of system design methodology. To study the huge volume of works and understand their contributions, we have categorized them into three broad categories: modeling, design, and analysis. We have compared and contrasted the MASIC approach with the others.

3. MASIC METHODOLOGY

MASIC—Maths to ASIC—is a system design methodology targeting signal processing applications. The three cornerstones of the methodology are the clean separation of the functionality from the architectural decisions, the systematic categorization of the architectural decisions, and the stepwise capturing of the architectural decisions through a number of abstraction levels. This chapter presents all these aspects of the methodology and discusses its benefits.

3.1 Introduction

As we move from algorithm on a chip to system on a chip era, more and more functionalities are integrated on a system. The present state-of-the-art systems contain a wide variety of components, which are connected through various types of communication architectures. As a result, the design and validation bottleneck is shifting from the implementation of the individual *digital signal processing* (DSP) functions to the *GLObal Control, Configuration, and Timing* (GLOCCT) that composes a system from these DSP functions. The GLOCCT is inherently tied to the system level architecture – processor cores, intellectual property (IP) blocks, buses, memories, and interfaces. In fact, the GLOCCT embodies the communication among different functional blocks of the system, and between the system and the environment.

The complexity of implementing and verifying the individual DSP functions has not decreased over the years. However, implementing the functional blocks has become a well researched, documented, and disseminated body of knowledge. Logic synthesis tools have significantly eased the implementation of these blocks. IP's are another useful design component that helps to cope with the complexities of the individual functional blocks.

To gain flexibility, support multiple standards, and adapt to changing requirements specialized systems often include a software part in conjunction with the hardware part. As a result, state-of-the-art systems typically embrace a complex architecture, which include a heterogeneous mix of hardware and software components, hundreds of types of signals, different segments of buses employing different protocols, and bridges among these different bus segments. Figure 19 shows a high level view of a SOC. Here, we focus on the design of the digital part of the SOC architecture, which is shown with a gray shade in Figure 19. As the number of functional blocks increases, the interaction and global resource sharing among them often increases in a geometrical way. Hence, the complexity of the GLOCCT increases rapidly, and the design and validation of the GLOCCT become more critical. Configurability, yet another factor that adds up to the complexity of the GLOCCT, is a key requirement for these large systems. Configurability is necessary to recover the huge investment made in the design process of a product, and hedge against the changing standards and performance requirements.

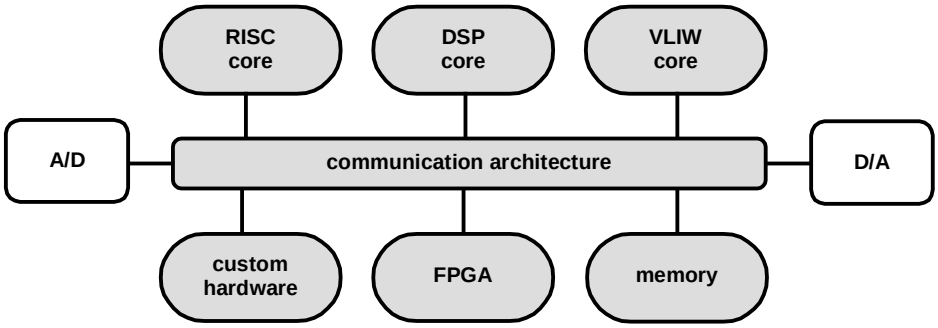

Figure 19. High level view of a SOC

On top of the design complexities the system designers face strict performance requirements. Embedded computing applications are different from most parallel processing systems developed for scientific computing or database management. Two major requirements differentiate the multiprocessor system-on-chip (MPSOC) design problems from the traditional parallel processing systems [135]:

- *Real time operation*: real time operation requires predictable performance. This does not mean that the architecture has to be trivial, such as eliminating caches, etc. It does mean that the architecture

behaves predictably enough so that the compiler and programmer can plan how to achieve the required computation rates.

- *Low power/energy operation*: power and energy constraints must be tackled at every levels of abstraction. The architecture's power/energy consumption characteristic must be as predictable as possible.

While the system complexity and performance requirements increase, growing time-to-market pressure demands these complex systems to be designed and verified with a lesser amount of time. This in turn makes the system design task more challenging.

The conventional register transfer (RT) level description, let alone the gate level description, is too detailed and not efficient enough to cope with the system design complexities. There is clearly a need of a higher level of abstraction when we attempt to model a complex system and explore the design space with different implementation alternatives. The MASIC methodology effectively tackles the challenges of designing complex systems. Before we present the MASIC methodology, let us briefly discuss the prevailing design methodology and the problems it faces in coping with the design complexities.

3.1.1 Prevailing Design Methodology

The practice in industry is to start at the functional level. At this level, algorithm development and functional verification are done. The implementation considerations taken into account at this stage are concerned with optimal implementation of individual DSP functions, like filter structure, order, etc. At this level, MATLAB is a popular tool. Later, the bit true simulators, like DSP Station, COSSAP or SPW are employed to model the functionality with finite word length effect. It is necessary to point out that these models do not capture the GLOCCT, which is present in the final product.

The next step is to make some chip level architectural decisions like:

- *I/Os*: how the system interacts with the surrounding functionality and how the start up sequence involving loading coefficients, gain parameters etc. are going to be achieved.

- *Type and number of execution resources*: number of DSP processor cores and hardware blocks, and division of functionality among them.

- *Global storage and interconnect*: number of busses and buffers.

These architectural decisions and the specification of the signal processing solution are then passed on to the RT level VHDL/Verilog design team in plain natural language. The RTL design team then implements the signal processing solution in the framework imposed by the architectural decisions. At this stage, the design team relies heavily on logic synthesis and module generators for implementation and simulation, and static timing analysis and equivalence checking as verification tools. The resulting gate level netlist is then passed on to the physical design team. A few iterations are often required between the logical and physical design flow before the design can be taped out.

Though researchers have developed clever solutions, the industry is slow to accept them, primarily due to the lack of interaction and controllability of these approaches. Moreover, for a system designer, architectural decisions are almost always evolutionary steps from the previous generation of the system, and are constrained by the surrounding functionality of the already existing system components.

3.1.2 Problems with Prevailing Design Methodology

The design methodology discussed in section 3.1.1 has four major problems:

- *Problem 1*: The functional modeling phase does not have any notion of time and it is not possible to validate the GLOCCT at that level. The GLOCCT enters the design flow at the RT level, which is very late and too detailed for any changes to be efficiently handled against the tight time schedule of the project.

- *Problem 2*: At the RT level, control and dataflow easily gets mixed up. In industry it is often desirable to keep these two separate because the dataflow represents the functionality (e.g., a wireless network basestation) of a system, whereas the GLOCCT embodies the implementation strategy that changes with different generations and families of products. For instance, the same dataflow in the first generation is implemented fully in hardware and has an individual datapath for each channel/data stream. In the next generation, the same dataflow could be implemented partly in software but still requiring separate resources for each channel. In the third generation, the technology could make it possible to have the same resource,

possibly all hardware, that can time multiplex the processing of all the channels. In all of these three cases the dataflow remains the same, but the communication architecture—hence the GLOCCT—changes.

- *Problem 3*: The hardware/software co-verification tools come into play at a later stage when a lot of the detailed hardware/software design has been already done by the design team. As a result, the designers get poor choices in partitioning between hardware and software, or selecting a different core.

- *Problem 4*: The architectural decisions and the signal processing specifications are conveyed to the RTL design team in plain natural language. Since the architectural decisions are not captured using any executable or synthesizable model, they can not be used as active ingredients in developing the RTL design.

MASIC squarely addresses the above problems. The following sections elaborately describe different aspects of the methodology.

3.2 Separation of Functionality and Architectural Decisions

On one hand, signal processing applications are data oriented where control is reduced to the extent of absence or presence of data. A bus interface is, on the other hand, control oriented where data handling is rather trivial an operation, for example copying or emptying buffers. When we attempt to realize a system, we find an intricate combination of dataflow and control flow in the final implementation. The system complexity, which exists in the final implementation, can be attributed to both of these contributing factors. A rational engineering approach to manage the design complexities is to break the sources of complexities apart.

If we look at the dataflow networks discussed in section 2.2.3, we find a number of concurrently executing processes. These processes perform some computation, and communicate with each other. In those abstract models, however, the inter-process communication is modeled in a simple way. We use point-to-point and theoretically unbounded FIFO channels to model the communication among the processes. In the final implementation of the system these abstract FIFO channels are typically implemented using memories, and a shared communication medium

is employed between the communicating parties. The communication and synchronization mechanisms depend on the implementation architecture. If we build the system models in such a way that the implementation aspects are separated from the system functionality then the designer gets the opportunity to concentrate on only one aspect of the design.

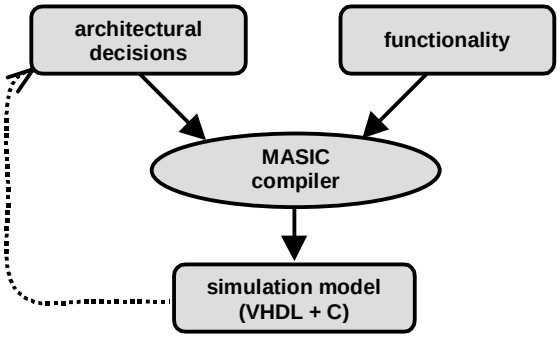

Figure 20. The basis of the system design flow in the MASIC methodology

The system design flow in the MASIC methodology is based on the principle of clean separation between the system functionality and the implementation architecture. Figure 20 shows the basis of the system design flow with the MASIC methodology, which is introduced in [35][70][72]. In this design flow, the functionality is modeled as abstract processes without any concern of how the processes would be implemented. The architectural decisions capture the information about how the computations within the processes are implemented, and how the processes communicate with each other. Finally, the functionality is mapped on the architecture to validate that the given functionality runs well on the chosen architecture, and meets the performance requirements. If the performance requirements are not met, the architectural decisions are changed, which is shown by the dotted arrow in Figure 20. The separation between the functionality and the architectural decisions splits the design complexity into two orthogonal axes, namely communication and computation. It also facilitates systems to be modeled at higher levels of abstraction, and reuse of predesigned and preverified cores using their interface description. In addition, it allows the two major aspects of any system to be developed separately, which is highly desirable in industry to meet the time-to-market constraints.

In the context of the separation between the functionality and the architecture, MASIC has quite a similar approach to a number of other contemporary system

design methodologies [44][83][84][102][122][131], which we have discussed in chapter 2. Keutzer et al. have elaborately discussed the separation between these two design aspects [83]. They have explained that the function is an abstract view of the behavior of the system, and characterized by its input-output relation. The implementation architecture, on the other hand, states how the function is implemented. Separation between these two aspects splits the design task into smaller and more manageable problems, which in turn helps to explore the design space with alternative solutions. Their ideas are commercialized in the Cadence *Virtual Component Codesign* (VCC) tool.

3.2.1 Functionality

Signal processing systems are often organized in a globally pipelined way. A sampled stream of data is buffered in a vector storage, processed by a DSP block, and streamed out for possibly further processing by similar combinations of storage and arithmetic operations. Besides sampled data, DSP functions often take some coefficients, and require some working buffer as shown in Figure 21. The arithmetic blocks, shown in the figure, perform certain computation in accordance with the given signal processing algorithm. The granularity of these functional blocks can be atomic operations, like adders or multipliers, or nonatomic operations, like filtering.

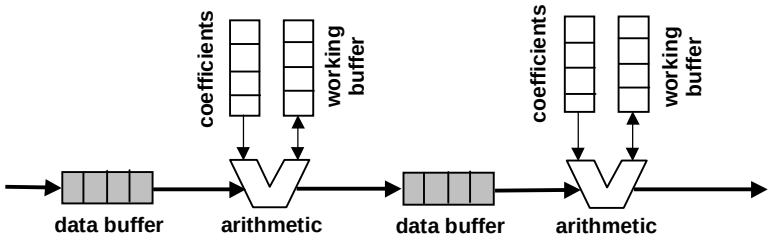

Figure 21. A typical signal processing system

These functional blocks and communication channels can be implemented in various ways. No matter how the functional blocks or the channels are implemented, the system must behave in the same way. This abstract behavior of the system is captured in the functional model. That is, it captures what the system is supposed to do, without specifying how it is implemented. Since signal processing systems are data dominated the functionality of these systems are commonly described using different dataflow networks, which we have discussed

in chapter 2. These formalisms effectively serve our purpose as well because they model the functionality without any implementation detail.

System modeling with MASIC begins at the functional level, which does not have a notion of time. The functional blocks perform computation in conceptually zero-time. The functional level modeling in the MASIC methodology has the same objective as the existing methodology described in section 3.1.1. At this stage, design issues are primarily algorithmic. The implementation considerations taken into account are concerned with optimal implementation of individual DSP functions, like filter length, coefficients, stability, etc.

The output of the functional modeling phase is a set of DSP functions in C that do not have any side effects. These functions do not have any notion of time and any means of exchanging data among them. Next, we build the GLOCCT to compose a whole system using the C functions developed at this level. The GLOCCT is responsible to provide the input data to the C functions, and fire the C functions to produce results at the output stream. Here, the advantage of the methodology is that we reuse the C functions developed at the functional phase to create a high level system model.

3.2.2 Architectural Decisions

The architectural decisions include the following issues:

- the interface to the environment, which defines how the system interacts with the surroundings
- the start-up sequence, which includes the reset, initialization of status signals, downloading of configuration data to configure DSP blocks, etc.
- the number and type of computational resources, like processor cores, IP blocks, or custom hardware
- the number and type of storage elements, like memories, buffers, etc.
- the number and type of interconnection mediums, like buses or on-chip networks
- the mapping of functionality on computational resources

- the movement of data between the computational resources and the storage elements

The architectural decisions tell us how the computation within the processes of the functional model would be implemented. That is, they specify how the functional blocks, which are represented using a piece of C code, is realized. The mapping of the functionality on an architecture is dictated by a number of factors, like performance, cost, power consumption, etc. While the performance requirements and constraints on power consumption call for hardware implementation of certain parts of the functionality, to gain flexibility and programmability specialized systems often contain a software part running on programmable cores. This is further aided by the fact that shrinking geometry and accompanying increase in performance in many instances allow functions to be implemented in software that earlier required implementation in hardware.

Signal processing systems are typically memory intensive. As a result, along with the implementation of the functional blocks using different hardware/software combinations, the implementation of the communication channels is an important factor as well. The architectural decisions also specify how the data FIFOs between the functional blocks are implemented, and how the interconnection topology look like.

Though the systems we consider are data dominated they often include reactive control as well, for instance to model the user interface, mode of operation, configurability, etc. For example, the I/O ports used to read input data during regular data processing may as well be used to download some filter coefficients during startup time. A typical signal processing system such as a wireless network base station is gradually accumulating complex control to support different configurations, product versions, standards, maintenance functions, fault tolerance, etc. Apart from the computation within the functional blocks and the communication between different functional blocks of the system, the GLOCCT also captures the communication between the system and the environment.

For the moment, let us assume that the architectural decisions are made by the designer based on his/her experience and using the design solution from the previous generation as a reference[9]. Irrespective of how the architectural decisions are made, the essence of the methodology is to capture them early in the design flow, whereas the established design methodologies, discussed earlier in section 3.1.1, capture the architectural decisions as late as the RT level.

[9] In section 3.3, we shall discuss about the architectural decisions in greater detail, and present a systematic approach to categorize them.

At this point, the prime benefit of the methodology is that the GLOCCT enters the design flow at a higher abstraction level than the RT level. Again, unlike the more conventional approach, the architectural decisions are no longer captured in plain natural language. We capture the decisions using the grammar based language of MASIC. In chapter 4, we shall discuss the grammar based language of MASIC that we use to describe the GLOCCT, and to build the system models.

3.2.3 The Abstract Model

The functionality and the architectural decisions are mapped together to form an abstract system model. Figure 22 shows the abstract view of the system. The system model consists of a number of concurrently executing blocks. The model is built at a higher level of abstraction, where the functional blocks are connected through point-to-point communication channels, and exchange data using atomic bulk transfer. The communication and synchronization among the individual functional blocks in the abstract model do not take place at the RT level of accuracy, instead abstract interfaces and abstract channels are used to exchange data. These communication channels are an abstraction of the actual pins and wires present at the implementation architecture. Since these communication channels avoid the intricate signaling protocol, developing a system model in MASIC and exploring the design space with different alternative design decisions becomes easier.

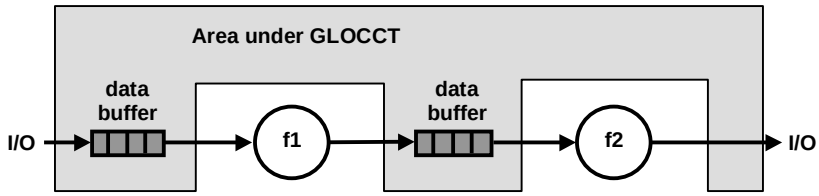

Figure 22. Abstract system model

The execution of each of the functional blocks is expressed using a piece of C code, which is developed during the functional modeling phase. However, the concurrency and the communication protocol of the system model cannot be expressed in a natural way using an inherently sequential language like C. Grammars, on the other hand, provide a natural and abstract way of describing

communication protocols[10]. Therefore, we describe the GLOCCT using the grammar based language of MASIC. The MASIC compiler generates a high level simulation model of the system in VHDL, as shown in Figure 20. The VHDL model imports the DSP functions in C, and fires them to perform a cosimulation using the Foreign Language Interface (FLI) of VHDL. The simulation results obtained from this abstract model can be used to dimension the architectural decisions.

The separation of the architectural decisions from the system functionality, discussed in this section, makes the modeling task easier. However, there exists a huge variety of architectural possibilities to implement a given functionality, and it is difficult and inefficient to consider them all at once. In the following section we discuss a systematic approach to categorize the architectural decisions that eases the system modeling task.

3.3 Categorization of Architectural Decisions

The system design flow shown in Figure 20 is referred to as the *Y-chart*—due to its distinctive Y-shape—in contemporary literatures [44][84][102][131]. To model a system using this design flow, a system designer typically studies the functionality, takes the system specification, makes a few initial calculations, and proposes an architecture to implement the functionality. Then the system performance is evaluated, for instance by simulation [84][131], and architectural decisions are altered to meet the performance goals. This approach suffers from a number of difficulties:

- There exists a huge number of architectural alternatives to implement both the computation and the communication aspects of a system. Exploring all the architectural decisions is an inefficient approach. Hence there is a need to categorize the set of architectural decisions in a systematic way, such that only a subset of the decisions needs to be considered at any particular instance in the design flow.

- There is a top-down iteration in the design flow shown in Figure 20. As a result, if a given set of architectural decisions does not yield a satisfactory performance then the design cycle needs to be iterated for another set of decisions. This long top-down iteration hinders the

[10] In chapter 4, we shall discuss more on the basis of opting for a grammar based specification technique.

design productivity. Hence, there is a clear need of creating intermediate levels of abstraction such that the top-down iteration is broken into smaller loops.

To alleviate the *Y-chart* based system design flow from the aforesaid shortcomings our objective is to categorize the architectural decisions and to break the long top-down iteration into smaller loops. For the purpose at hand, we take a closer look at different formalisms used to model signal processing systems, and present a systematic approach to classify the architectural decisions in two broad categories: ***system level decisions*** (SLDs) and ***implementation level decisions*** (ILDs) [41]. The SLDs form the class of decisions where the designer has less freedom to explore the design space because these decisions are dictated by the system interface, the available computational resources, and a schedule for an optimum implementation of the processes on the resources. In section 3.3.1 we shall elaborately discuss the SLDs, and show that they in fact capture the design decisions along the computation axis of a system model. The ILDs, on the other hand, capture the design decisions along the communication axis of a model. Typically, a designer gets more freedom of exploring the ILDs than the SLDs.

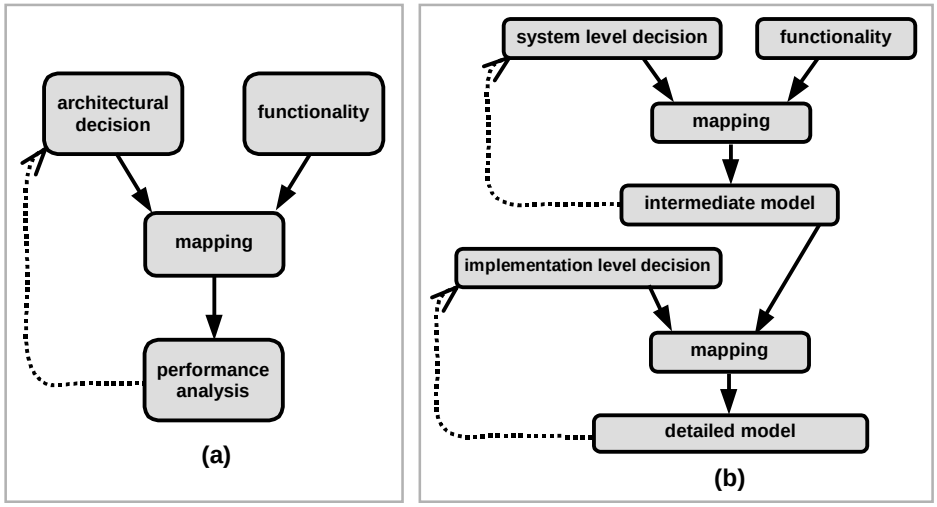

Figure 23. (a) Y-chart based system design flow, (b) Design flow after categorization of architectural decisions

Figure 23(b) shows that we add the SLDs to a functional model to create an abstract intermediate model. To build a more detailed system model, the ILDs are

added to the intermediate one. The formulation of the intermediate model breaks the top-down iteration, which exists in the *Y-chart* based approach shown in Figure 23(a), into smaller loops. Next we discuss the basis of the categorization of the architectural decisions in more detail.

3.3.1 System Level Decisions (SLDs)

Functional modeling of signal processing applications usually begins using Kahn Process Networks (KPNs) [78], or different forms of dataflow networks like synchronous dataflow (SDF) [91][92], dynamic dataflow (DDF) [26], etc. These models capture system functionality without any architectural details and have become the most popular choice for building functional models. Commercial simulators [2][5] execute such models orders of magnitude faster than an RT level simulator. Processes in a KPN are connected through infinite length point-to-point FIFOs. Graphically, processes are drawn as nodes and FIFOs as arcs. Processes read from the input FIFO when data is available (i.e., blocking read), perform computation in their private memory, manipulate their own state space, and write results in an output FIFO (non-blocking write). This non-blocking write property in turn means that the depth of the FIFOs is theoretically unbounded.

While a function is mapped to SW to achieve flexibility, mapping to a HW block is done for performance critical parts. Traditionally, the mapping of functions on computational resources is viewed as a scheduling problem. Scheduling is an important issue regarding an optimum implementation of these networks that determines which process executes on which resource at which point in time. A highly cherished property of KPN is that the ordering of the execution of processes, that is the schedule, has no effect on the network behavior. In other words, given some tokens at the input FIFO, the network will produce the same output irrespective of the schedule.

Finding the FIFO size is not possible for the general case of KPN. However, there exist scheduling techniques to find a reasonable upper bound using simulations with varying input data set [16]. It is necessary to mention that the FIFO sizes determined in this way cannot guarantee a deadlock free execution. This is because a different set of input data than the ones used in the simulation may lead to an artificial deadlock, which would not have occurred if the lengths of the FIFOs were infinite.

Although the infinite FIFOs make functional modeling easier, it is not possible to realize them in practice. However, there exists an efficient technique of scheduling

SDF networks for a sequential (single processor) or parallel (multiprocessor) implementation [91]. The SDF scheduling technique solves the practical problem of finding the sizes of the FIFOs such that an iteration of process firing can be performed using bounded memory [26]. Firing of an SDF node consumes a fixed number of tokens from each input arc, and produces a fixed number of token on each output arc. The minimal set of firings, that bring the FIFOs to their initial size, defines an iteration. This is shown by a *balance equation* for each arc between the processes of a network. The balance equation for an arc between process *i* to process *j* is given by:

$$r_i p_i = r_j c_j \tag{3.1}$$

where process *i* produces p_i tokens on the arc, and process *j* consumes c_j tokens from the arc. The variables r_i and r_j show the number of firings of process *i* and process *j* respectively.

The order of process execution given by a schedule is not enough to build an intermediate system model where the execution of the functional blocks would be properly synchronized in accordance with different architectural limitations. This is because:

- Scheduling techniques, for example [91], only provides a sequence of process execution, where execution is an atomic operation.
- Scheduling assumes the ideal situation that processes have their own local memories. However, the processing element, on which the process is mapped to, may not have enough data memory. The architectural limitations of the processing elements call for more synchronization primitives on top of the order of process execution suggested by a schedule.
- Scheduling, like [91], does not deal with the system interface to the environment and buffering of input data. Hence, the assumption of being able to schedule an input node at *any time* has to be synchronized with the availability of valid data at system input.

The implication of the synchronization primitives to account for architectural limitations can be better explained with the help of a small example. Let us consider the example SDF network shown in Figure 24(a). Here process `p1` reads

one token from input FIFO `f1`, which has a delay[11] of 2 units; and produces one token on output FIFO `f2`, which has no delay. Corresponding values for the other processes and arcs are shown in the figure. Considering a run time of 1-unit for process `p1` and `p2`, and 3-units for process `p3`, an optimal schedule for the SDF network is shown in Figure 24(b). Here process `p1` and `p2` are running on processor1, and process `p3` is running parallelly on processor2. This schedule runs well if FIFO `f1` has a FIFO depth of two.

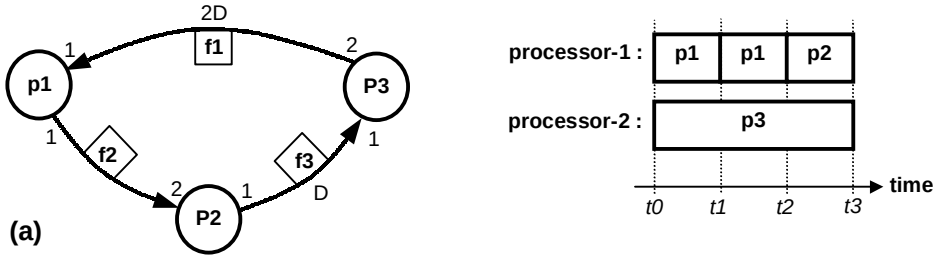

Figure 24. (a) An SDF network, (b) schedule for two parallel processors

Now, let us consider the implications of architectural limitations on this schedule. If processor2 does not have enough memory it has to write the result directly to the output FIFO `f1`. Since the size of `f1` is bounded, this operation would overwrite the FIFO and cause processor1 to read new data before it has finished processing the old data. To properly synchronize this write operation of processor2, it has to wait according to the architectural limitations of processor1:

- *If processor1 has enough memory to buffer input data*: Processor2 has to wait until processor1 has finished reading data from `f1`.
- *If processor1 does not have an input buffer*: In this case the data needs to be available in the FIFO until processor1 has finished executing the process `p1`. So, processor2 has to wait until processor1 has finished its execution.

When these synchronization primitives are added on top of the order of process execution given by a schedule then the resulting model captures the system behavior taking the architectural limitations into account.

[11] Here the term *delay* is used in the signal processing sense where a unit delay on an arc from process *p1* to *p2* means the n^{th} sample consumed by process *p2* is the $(n-1)^{th}$ sample produced by process *p1*.

From the preceding discussion we figure out that the SLDs consist of the following design decisions:

- process to resource[12] mapping
- schedule of process execution
- FIFO size
- synchronization primitives in accordance with the interface to the environment and the architecture of the resources

The process to resource mapping and their schedule tell us how the computation is performed, and basically capture the design decisions along the computation axis of a system model. This is the class of decisions where the designer has less freedom because they are dictated by the system interface, the available resources, and a schedule for an optimum implementation of the processes on the resources. If performance requirements are not met then these decisions are changed. In section 3.4, we shall discuss how the SLDs are added to the functionality to create an intermediate system model.

3.3.2 Implementation Level Decisions (ILDs)

The ILDs are the class of decisions where the designer has more freedom to explore the design space. The SLDs capture the process to resource mapping along with their execution schedule, and the sizes of the FIFOs. However, it does not capture any information regarding the implementation of the FIFOs and the communication architecture. Since the multiprocessor SOC (MPSOC) designs are typically memory intensive, the implementation of the memory system plays a significant role in determining the system performance. On top of that, the memory system is often a major determinant of energy consumption [135]. At this level we decide how the FIFOs are going to be implemented and how the communication architecture will look like.

There is a wide design space related to the implementation of the communication architecture. For example, the FIFOs could be implemented using message passing where the communication among processors takes place through multiple private memories, implying explicit communication by sends and receives. On the other

[12] Here, the word *resource* is used to refer to the computational resource, not the interconnection resource.

hand, processors may have a shared memory, implying implicit communication by loads and stores. Again, the single address space of the shared memory may have centralized or distributed physical memory. The interconnection medium may be realized with different types of bus architectures, the bus might have different widths, protocols and arbitration priorities, etc. Since the traditional bus based architectures do not scale well with increased number of system components, a Network-on-Chip (NOC) architecture, for example the one proposed in [87], can be used to implement the communication architecture.

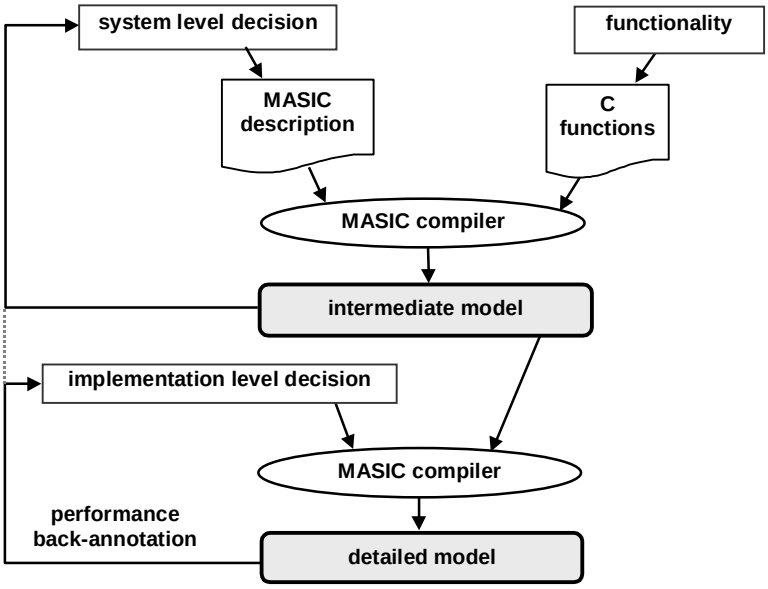

Figure 25. Design flow in MASIC after categorization of architectural decisions

Figure 25 shows the conceptual view of the design flow discussed in this section. The SLDs are captured using MASIC grammar notations, and the functional blocks are described in C. The MASIC compiler generates an intermediate system model in VHDL, and creates links to the C functions using the foreign language interface (FLI) of VHDL. The estimated computation time of the functional blocks is added in this model to simulate the system with computation delays. If the simulation of this model yields a satisfactory result the ILDs are added to the intermediate model to build a detailed model. This detailed model shows the cycle true transactions of the communication architecture. Communication architectures add significant amount of delay due to synchronization overhead [88]. Effects on

system performance due to shared memory access, especially when system operation is dominated by memory accesses, have been reported in [89]. Simulation of this detailed model reveals these effects. If the performance is unsatisfactory then the ILDs need to be changed, which only requires the design step from the intermediate model to the detailed model to be performed. Thus the top-down iteration of the initial design flow, shown in Figure 20, is broken into two smaller loops. However, if changing the ILDs alone are not enough to meet performance, then the initial mapping and scheduling needs to be changed. This is indicated by the dotted line in Figure 25.

3.4 Levels of Abstraction

In section 3.2 we have discussed about the separation of the system functionality from the architectural decisions. Next, in section 3.3, we have presented a systematic approach to categorize the architectural decisions. Now the question is how we are going to capture the architectural decisions, and map the functionality on the architecture that we have decided to realize.

The design flow from an abstract level to an implementation level can be considered as a sequence of steps that removes the freedom of choice to implement the abstract functional model. For instance, the SLDs capture a process to resource mapping decision. If it is found that a certain mapping does not yield satisfactory performance then the SLDs need to be changed. Changing the design decisions require a major redesign effort if the design decisions are added into the model with greater detail at lower abstraction level. To minimize the redesign effort, we need to map the functionality to the system architecture at higher abstraction level such that the system models can be created quickly, and simulated with faster speed. In addition, we need the abstract models to yield as accurate performance estimates as possible.

In the context of signal processing systems we have defined three ***transaction level models*** (TLMs), which capture the design decisions using abstract transactions [42]. These TLMs reside at different levels of abstraction between the functional and the implementation model of a DSP system. The basic intention behind the creation of the TLMs is to include the artifacts of the design decisions—without performing any detailed design work—into the model, such that the simulation of the model gives as accurate estimates as possible.

The concept of TLM first appeared in the domain of system modeling languages, like SystemC [59] and SpecC [48]. In [59], a TLM is defined as a model where

communication between modules is modeled in a way that is accurate in terms of what is being communicated, but not in a way that is structurally accurate (i.e., the actual wires and pins are not modeled). They have also insisted that the communication between modules of a system be modeled using function calls.

Gajski et al. have defined a TLM as a model where the details of communication among components are separated from the details of computation within the components [28]. Hence the unnecessary details of communication and computation are hidden in a TLM. They also present the *system modeling graph* where computation and communication are shown in two axes. Each axis has three degrees of time accuracy: untimed, estimated, and cycle accurate.

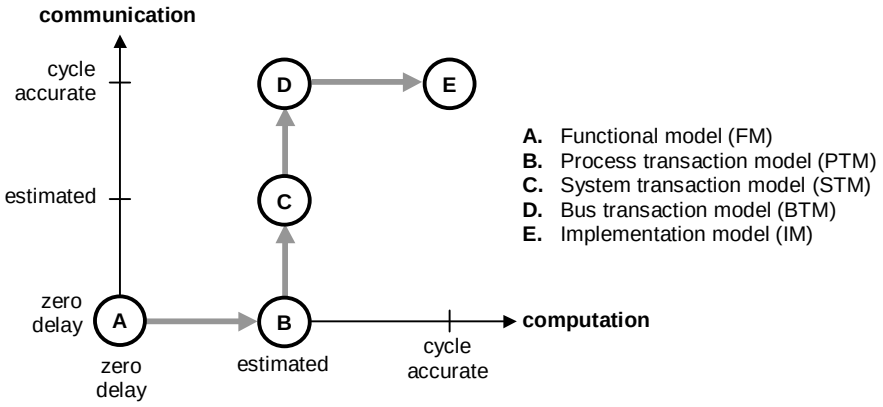

Figure 26. Coordinates of different system models in system modeling graph

Typically, the ***functional model*** (FM) of a DSP system is built at the most abstract level where both communication and computation takes place in zero-time. In the ***implementation model*** (IM), however, both communication and computation are cycle accurate. In the context of DSP systems design, we present three TLMs between the FM and IM. These TLMs are the ***process transaction model*** (PTM), the ***system transaction model*** (STM), and the ***bus transaction model*** (BTM). The coordinates of all these models are shown in the system modeling graph of Figure 26. The thick arrow in gray shows the course of modeling from the functional level to the implementation level. These intermediate TLMs capture different aspects of the system, and allow us to traverse the system modeling trajectory in small incremental steps.

3.4.1 Comparison of Different System Models

Before we go into the detail of the different system models at different abstraction levels, a comparison of the models is tabulated in Table 1 to help the reader to take a quick grasp of the different models.

Table 1. Comparison of different system models

	Communi-cation	**Computa-tion**	**Transaction/Operation**	**Model characteristic**
Functional Model (FM)	zero-delay	zero-delay	- **process execution** (atomic execution consisting of read, computation, and write operations)	Process to resource mapping is not done, communication through ***infinite*** length FIFO
Process Transaction Model (PTM)	zero-delay	estimated	- **bulk read from FIFO** - **computation** (C function call, with estimated delay) - **bulk write to FIFO**	Process to HW/SW mapping is assumed, communication through ***finite*** length FIFO, read/write to FIFO using get/put procedures
System Transaction Model (STM)	estimated	estimated	- **bulk read from FIFO** (with estimated delay) - **computation** (C function call, with estimated delay) - **bulk write to FIFO** (with estimated delay)	communication through get/put procedures from/to FIFO with estimated delay for memory access using shared medium
Bus Transaction Model (BTM)	Cycle accurate	estimated	- **memory read** (Req, Ack, Address, Data, Split) - **computation** (C function call, with estimated delay) - **memory write** (Req, Ack, Address, Data, Split)	communication through cycle accurate component interface and shared medium, read/write to memory using physical address
Implemen-tation Model (IM)	cycle accurate	cycle accurate	- **memory read** (Req, Ack, Address, Data, Split) - **cycle accurate computation** (RTL, ISS) - **memory write** (Req, Ack, Address, Data, Split)	cycle accurate computation– RT level for HW implementation, or instruction level for SW implementation

3.4.2 Functional Model (FM)

As shown in the system modeling graph of Figure 26, both communication and computation in the FM takes place in conceptually zero-time. This model resides

at the highest level of abstraction. At this level, individual functional blocks are viewed as processes acting on infinite stream of data. The processes perform some computations, and communicate with each other. These processes are connected by point-to-point FIFOs, and they have no other means to communicate with each other except these FIFO channels. The communication and the computation aspects of this model can be implemented in a wide variety of ways.

We develop the individual processes in C. The granularity of these processes can be atomic or nonatomic operations. At this stage, the design issues are primarily algorithmic and selecting different dataflow structures to realize individual functional blocks (e.g., selecting a direct-form or data-broadcast structure to realize a FIR filter, etc.). Here, verification is concerned with making sure that the specified signal processing figures of merits are met. The output of this level is a set of DSP functions in C, which does not have any side effect. These functions perform atomic execution – that is, reading data from the input FIFO, computing the result from the input sample, and writing the result to the output FIFO is considered as an indivisible operation.

3.4.3 Process Transaction Model (PTM)

The coordinate of PTM in the system modeling graph is shown in Figure 26. The communication among processes at the PTM level takes place without any delay, and the computation delay of the processes is at an estimated level. To create a PTM, we add the SLDs to the FM. However, no detailed design work is carried out to add the SLDs. For this purpose we look at the *demand-space*, that is the transactions that the processes of the application demands to make, and only capture the process-level transactions. At lower level of abstraction, these process-level transactions are elaborated to the *implementation-space*, i.e., the detailed signaling protocol of the implementation architecture.

Scheduling techniques, as those discussed in section 2.3.3, yield the order of atomic process execution over time. To build the PTM we split the atomic process execution into a sequence of process-level transactions—*read*, *compute* and *write*—and these operations are properly synchronized. The addition of these synchronization primitives keeps the order of process execution as given by a schedule. However, they add the necessary synchronization to make the PTM interact correctly with the environment, and to account for the architectural limitations of the resources, which were pointed out in section 3.3.1. To be able to successfully read data, typically a process needs to know if there are new data available in the input FIFO. After input data is read, the compute operation can be

performed using the C function developed at the FM level. To write the result of the computation safely, the process needs to know if there is room available in the output FIFO because after the addition of the SLDs, the FIFOs are no longer infinite.

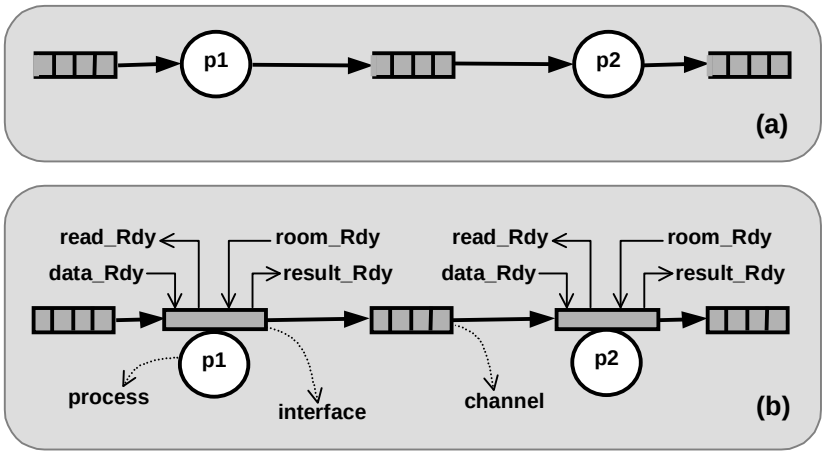

Figure 27. (a) Inter-process communication at FM level, (b) Inter-process communication at PTM level

Figure 27(a) shows the inter-process communication using infinite FIFOs at the FM level. The typical inter-process communication at the PTM level is shown in Figure 27(b) where the processes are connected through bounded FIFOs. An abstract system controller controls the dataflow between the processes such that the producer process must wait if the output FIFO is full, and the consumer process must wait if the input FIFO is empty. Figure 27(b) illustrates the fact that the dataflow through the channel is controlled by a couple of signals that captures the major synchronization events.

Here, the controller provides the `data_Rdy` and `room_Rdy` synchronization signals to processes. These signals, respectively, tell the process if there is data available in the input FIFO, and if there is room available in the output FIFO. The process outputs the `read_Rdy` and the `result_Rdy` signals, which respectively tell the controller that the process has finished reading the input data, and it finished its computation. Processes may perform read and write transactions over the dedicated FIFO channels using `get` and `put` procedures, when the `data_Rdy` and `room_Rdy` signals are asserted, respectively. The `get` and `put` procedures provide atomic bulk transfer capability and do not require any physical address or complex

handshaking protocol. Hence modeling at this level is fairly easy, and it still captures the timing to model the major synchronization events. The control flow of a typical process in PTM is shown in Figure 28. It leaves the `INIT` state when `data_Rdy` is available, then `gets` the data, emits the `read_Rdy` signal, computes the result, `waits` for an estimated time to model the computation delay, emits the `result_Rdy` signal, puts the result when the `room_Rdy` is available, and finally loops back to read more data when `data_Rdy` is available.

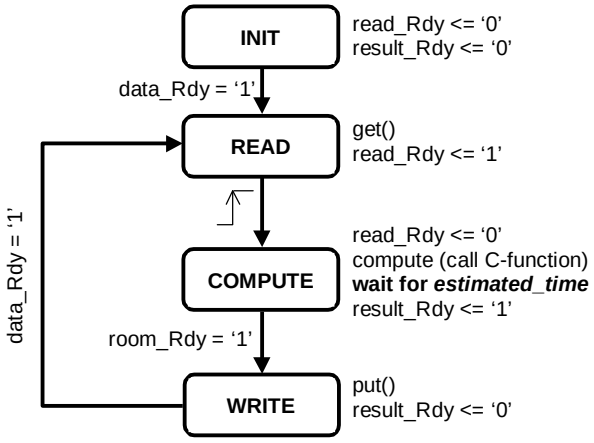

Figure 28. Control flow of a typical process in PTM

In the next chapter we shall discuss how the PTM is developed using the MASIC language. Though the computation is done in zero simulation time using a C function, the estimated computation delay is included in the model using `wait` statements. As a result, the simulation of this model shows the system performance based on the estimated computation delay. If the run-time of a function is data independent, the computation delay of the function on a target architecture can be found using system level estimators. Even for the data dependent case, the computation time is bounded in hard real time applications[13].

If the simulation of the model does not yield satisfactory performance then the SLDs are changed. For example, the number and type of resources can be

[13] A *hard real time system* guarantees that critical tasks complete on time [124]. This guarantee requires that all delays in the system be bounded. As a result, advanced operating system features, e.g., virtual memory, are avoided in such systems. A less restrictive type of real time system is a *soft real time system*, where critical tasks get priority over other tasks. Given the lack of deadline support, soft real time systems are risky to use in safety critical systems, industrial control, etc.

augmented to add computational power. Again, if more memory is available, the sizes of the FIFOs can be increased keeping the relative proportion of their initial sizes [16]. Such an increase in the FIFO size would cause less context switches. If relatively little computation is done each time a process is executed, the reduction in context switches yields a better schedule with higher performance.

It is necessary to note that, though a process to resource mapping is assumed no detailed design work is carried out to capture the SLDs. Hence, it does not require any tedious redesign effort to change the SLDs at the PTM level, and simulate it with another set of decisions.

3.4.4 System Transaction Model (STM)

While the PTM captures the process to processor mapping and the computation delay, it does not capture any information regarding the implementation of the FIFOs. At this level, we decide how the communication architecture would be implemented, that is we consider the ILDs. However, no detailed design work is carried out. We only add the estimated communication delay in the model.

Communication architectures add significant amount of delay due to memory access through shared medium and synchronization overhead [88][89]. The delays depend on bus width, bus protocol, priorities, DMA block size, etc. Currently, we estimate the communication delay by observing the communication protocol of the target bus architecture in an ad-hoc way. However, more accurate delay figures can be estimated, for example by using the trace based analysis technique presented in [88]. Irrespective of how the delay is estimated, the purpose of the STM is to capture the delay at an abstract level.

STM is truly the system model that includes information on both the computation and the communication aspects of the design, though both of them are at an estimated level. The transactions modeled in an STM are the same like the ones in PTM, except that it `waits` after each `get` and `put` transaction to model the communication delay. At this level, processes still communicate through dedicated point-to-point channels. The abstract controller provides the `data_Rdy` and the `room_Rdy` signals.

The control flow of a typical process in an STM is shown in Figure 29. It also leaves the `INIT` state when `data_Rdy` is available, and then `gets` the data. Unlike the PTM, after getting the data it `waits` for an estimated amount of time before emitting the `read_Rdy` signal. Then it computes the result, `waits` for an estimated time to model the computation delay, emits the `result_Rdy` signal, puts the result

when the `room_Rdy` is available, and again, unlike the PTM, it `waits` for an estimated amount of time before deasserting the `result_Rdy` signal. Hence, the simulation of this model shows both the communication and the computation delays at an estimated level. If simulation result of the STM is not satisfactory then we need to change the ILDs. Since system transactions have not yet been modeled using the actual wires and pins, changing design decisions is fairly easy at this level.

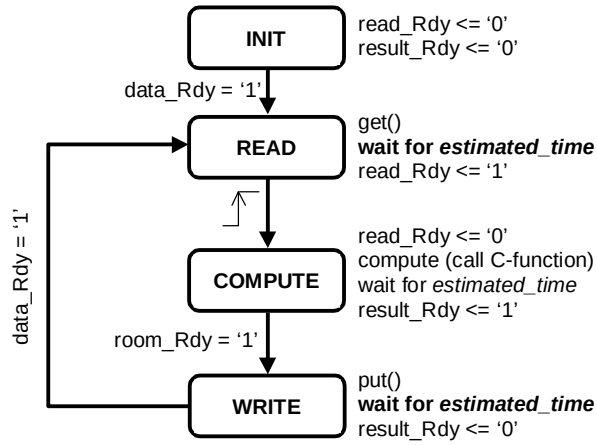

Figure 29. Control flow of a typical process in STM

The prime purpose of building an STM is to validate that a given functionality runs well on the chosen architecture, and meets the performance requirements. On one hand, it is desirable to build system models at higher levels of abstraction. On the other hand, the higher abstraction level the less accurate the performance estimates are. However, our experimental results show that an STM can be created and simulated much faster than a lower level model, and the simulation results are reasonably accurate compared to the clock-true level simulation results. Thus the STM effectively helps the designer to explore the design space along both the computation and communication axes.

3.4.5 Bus Transaction Model (BTM)

In STM, we have captured the SLDs and ILDs using abstract transactions. Now, at the BTM level we model the component interfaces, the communication

architecture, and the transactions among the resources at the structurally accurate level. After the transaction, when the data is available in the component, the computation operation is performed by a C function call.

We build the cycle-true interfaces of functional blocks into which the functionality is mapped to. The interface represents the structurally accurate interface of an IP block, or the *bus functional model* (BFM) of a predesigned programmable processor core. The BFM of a core represents the external interface of the core and generates the cycle accurate transactions supported by the core (e.g., read, write, burst read, burst write, etc.) to the system. The interface may as well describe the interface of a custom block, which we have decided to design to implement a performance critical functionality. The interface descriptions of the predesigned blocks are saved in a library, which effectively helps the designer to achieve design reuse.

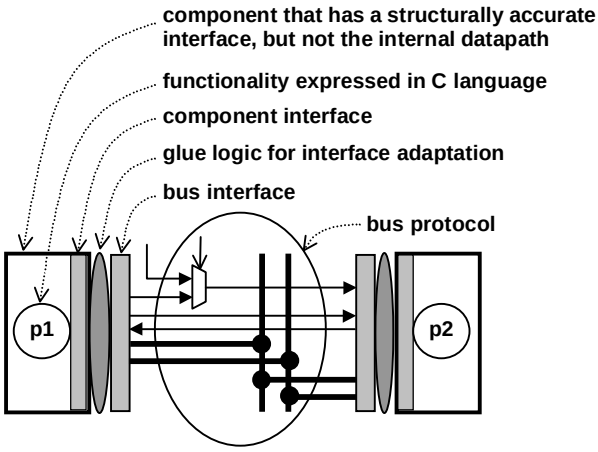

Figure 30. Communication architecture is structurally accurate at the BTM level

The atomic bulk transfers between the synchronization points of the STM are now elaborated using the interface signals of the functional blocks. These blocks access the memory through a shared communication medium (e.g., bus or network) using the communication protocol and the physical address. That is, the abstract `get` and `put` operations are spread over clock cycles using the actual signals, like `Req`, `Ack`, `Data`, `Address`, `Split`, etc. The BTM does not alter the execution semantics of the STM because the points of synchronization of the STM are still maintained. Often it happens that the functional blocks and the communication architecture do not

use a compatible protocol. As a result, we also need to develop the glue logic between the functional block and the communication architecture.

Figure 30 shows a typical system model at the BTM level, where the components communicate using the actual signals. Though the component models have a structurally accurate interface, they do not have their internal datapath. Once the data is transferred to these components, a piece of C code is executed on the data to compute the result. Simulation of the BTM reveals the clock true communication delay of the implementation architecture. The design step to create a BTM from an STM involves the following tasks:

- Elaborating the abstract communication channels of the STM into the detailed signaling mechanism required to express the bus protocol and the arbitration logic of the implementation architecture. If several functions are mapped on a single core, the channel between the functions needs to be implemented using the communication primitive offered by the operating system or the RTOS kernel.

- Describing the interface of the hardware blocks onto which the functions are mapped to. To ease reuse of predesigned components we save the BFMs of embedded cores and interface description of IP blocks in a library. MASIC descriptions are used to build these models and they are instantiated from the library to build the BTM of a system.

- Adapting the component interfaces to the protocol of the communication architecture when they employ incompatible communication protocols. We describe the glue logic between the predesigned blocks and the architecture using the MASIC language. These interfacing adapters can be saved in the library and reused afterward. Currently these adapters are written manually. However, this task can be automated, for example, using the approach presented in [109].

The design flow to create a BTM is given in Figure 31. This is in fact the three stage unfolded view of the initial design flow shown in Figure 20. Figure 31 shows that the PTM captures the system functionality and the SLDs, where the SLDs are described using abstract transactions. While the functionality is expressed in C, the SLDs are expressed using the grammar based MASIC language. Next, to create the STM, we add the ILDs using abstract transactions. If the STM simulation results are satisfactory, these ILDs are then described at the clock-true level to create the BTM. MASIC descriptions are used to create the

library of component interfaces. These structurally accurate component interfaces are instantiated in the BTM from the library.

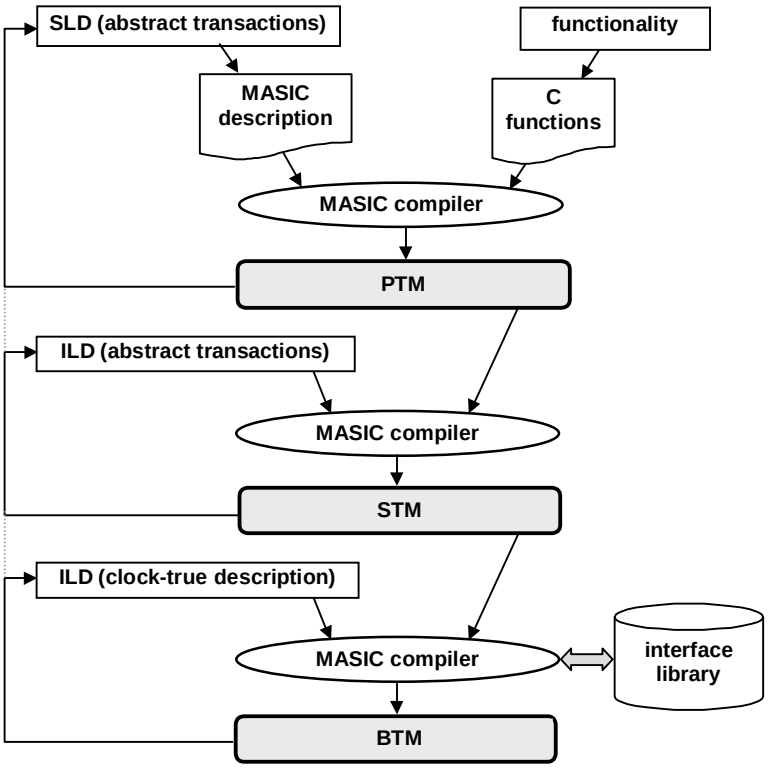

Figure 31. Design flow to create the BTM

3.4.6 Implementation Model (IM)

As shown in the system modeling graph ofFigure 26, both the communication and the computation are clock true at the IM level. At this level, the computation operation is no longer performed through a C function call. The SLDs have already captured a process to resource mapping decision, and we have included them at the PTM level using abstract transactions. Based on the SLDs, there are a number of ways to realize the C functions. For HW implementation, there are two different possibilities. If a function is mapped to a predesigned IP block, we need

to replace the interface of this block with the RT level descriptions of the component. If we decide to realize the functionality on a custom hardware, then the simulation results of the BTM helps the designer by providing the constraints on the interface of the module to be synthesized. For a SW implementation, on the other hand, the C functions are compiled to the target architecture.

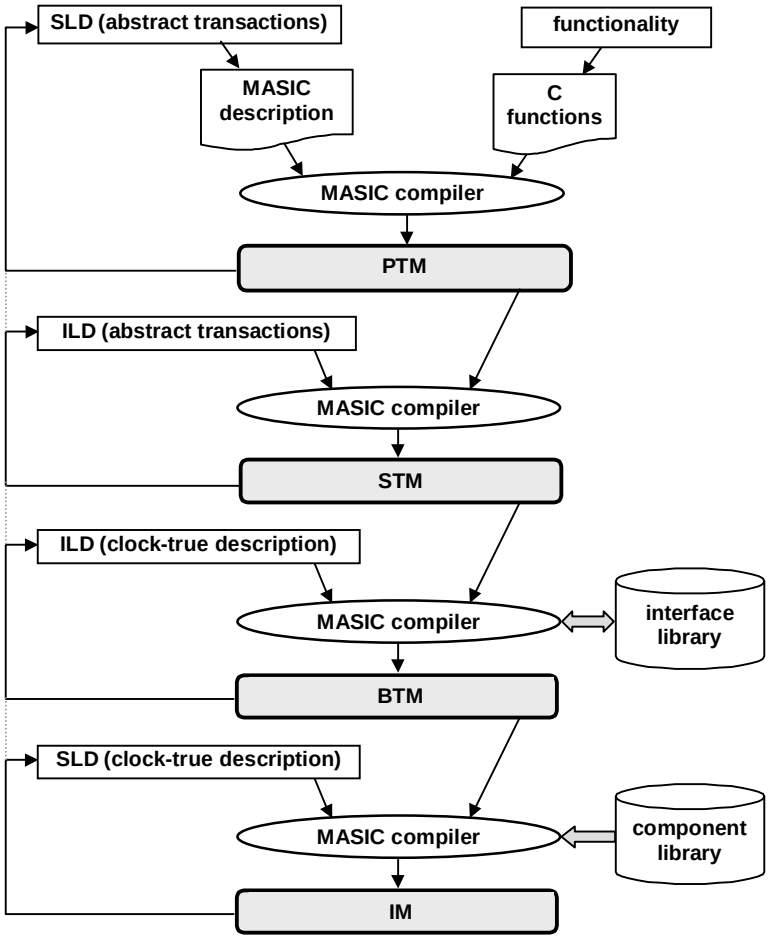

Figure 32. Design flow to create the IM

The glue logic at the BTM level was developed taking the cycle accurate component interface (e.g., BFM), and the communication protocol into account.

As a result, the glue logic developed at the BTM level can be reused at the IM level.

Since the component interface models of the BTM are replaced with the real components with internal datapath, the computation operation now takes place on the actual datapath of the resource onto which we have decided to map the functionality. That is, for a hardware implementation the computation takes place at the RT level, whereas for a programmable core based implementation the computation takes place at the instruction set level. The simulation results at this level show the clock-true delay due to both the computation and the communication.

Figure 32 shows the design flow to create an IM. The component library at this stage contains RT level components with internal datapath. These components are not created by the grammar based language of MASIC. They have been developed using the RT level VHDL description.

3.4.7 Design Flow through Different TLMs

The overall system design flow through different TLMs is shown in Figure 32. It is of course possible to go directly to the IM level from the FM level. In this case, the SLDs and the ILDs need to be described elaborately in a single step. This clearly involves a huge design effort, and incurs a long design time. If, at the end, the detailed model does not meet the performance goals then an expensive redesign effort becomes inevitable. The purpose of creating the intermediate TLMs is to formally capture different aspects of a system in a systematic way. As a result the long design effort can be broken into short incremental design steps, the intermediate models can be checked against performance requirements at different abstraction levels, and the design decisions can be changed if needed.

However, in certain cases, the creation of a number of intermediate models may not be so desirable to the designer. If that is the case, then a designer can combine some of the steps. For example, Figure 33 shows a design flow that directly moves from the FM level to the STM level, and skips the creation of the PTM. This can be done by simultaneously capturing both the SLDs and ILDs using abstract transactions, which yields the STM with estimated computation and communication delays. From the STM, a designer can move directly to the IM, or take the path through the BTM.

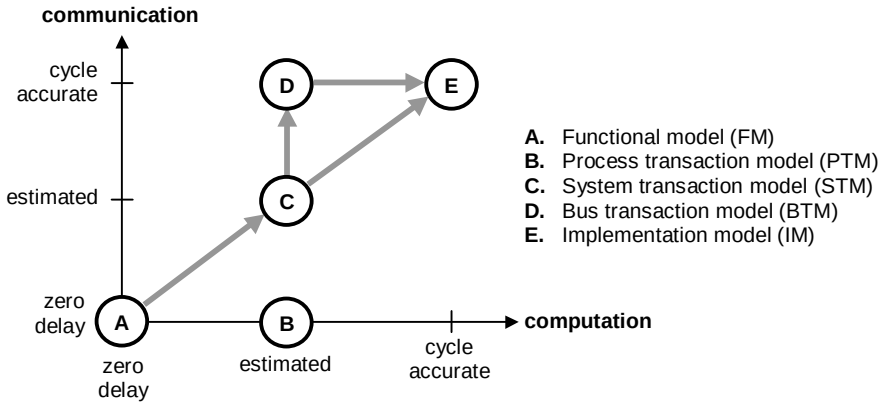

Figure 33. Combining design steps

It is also possible to follow the design flow shown in Figure 34, where we have shown a model at the point `F` of the system modeling graph. At this point, the computation is cycle accurate but the communication is estimated. Here, components with structurally accurate interface and datapath are connected to abstract communication channels via bus wrappers. This design flow can effectively benefit from the component based design methodologies. The component based design methodologies, for example [30], provide automatic wrapper generation tool to synthesizes hardware interfaces, device drivers, and operating systems.

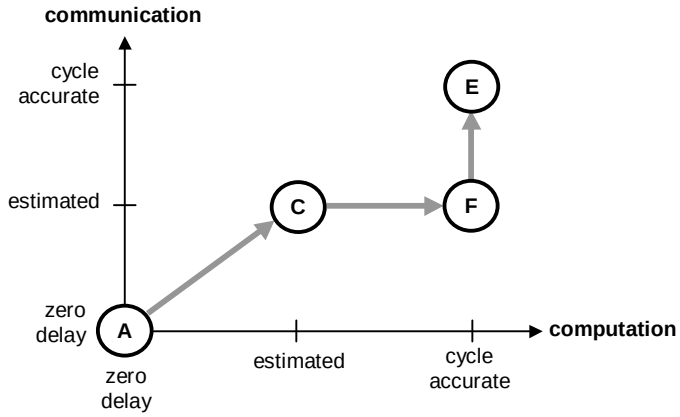

Figure 34. Design flow using component based design style

3.5 Conclusion

In this chapter we have presented the MASIC methodology for signal processing systems design. MASIC cleanly separates the functionality and architectural decisions. Consequently, the design complexity is split into two orthogonal axes, which helps the designer to cope with design challenges. The architectural decisions, regarding the implementation of the GLOCCT, enter the design flow much earlier than the detailed RT level. We have presented a systematic approach to categorize the architectural decisions such that the designer does not need to consider all the possible alternatives at once. The categorization of the design decisions allows us to create intermediate models at different levels of abstraction, which in turn eliminates the top-down iteration present in more conventional design flows.

Mapping of the functionality on a heterogeneous architecture is a nontrivial task. If the initial design decisions do not yield satisfactory performance, then changing the decisions would require a major redesign effort if they are captured with greater detail at a lower level of abstraction. To minimize the expensive redesign effort we have defined three TLMs, which reside at different levels of abstraction between the functional and the implementation model of a DSP system. The formulation of these TLMs eases the design task of realizing an implementation model from an abstract functional model and the exploration of the design space at a higher level of abstraction. In addition, the design flow effectively facilitate the reuse of predesigned cores and HW blocks using their interface descriptions. In summary, MASIC offers a smooth path from the functional modeling phase to the implementation level, facilitates the reuse of HW and SW components, and enjoys existing tool support at the backend.

Currently the interfacing adapters are written manually. However, this task can be automated using the ideas presented in [109]. Currently the communication delay is not estimated using any rigorous technique. The simulation accuracy of the STM can be improved by using the delay figures found from the trace based analysis technique shown in [88].

Although the TLMs are meant to formally capture different aspects of a system and help to explore the design space in a systematic way, the creation of a number of TLMs may tend to be not always welcome by a designer. In such a case, as discussed in section 3.4.7, a designer can skip creating a TLM and move to the next level of abstraction. However, the redesign effort to change the design decisions to meet performance would be higher in such a case.

The methodology, as it stands now, only considers the consequence of a design decision in terms of computation and communication delay. Though delay is an important factor for the kind of systems that we are interested in, the power consumption is also a significant issue. In the future it is needed to incorporate the power estimates in the TLMs so that the design decisions can be explored in accordance with power/energy constraints.

To map the functionality on system architecture, we have relied on design decisions. Another way of moving to an implementation level is to employ semantic preserving transformations [117]. Semantic preserving transformations do not change the meaning of the model, and are mainly used for design optimization during synthesis. Design decisions, on the other hand, may change the behavior of the model. For example, the PTM with a finite sized FIFO may not behave in the same way as the functional model does when an artificial deadlock occurs, which would not have happened if the FIFOs were infinite. Since infinite FIFOs are not practically realizable we need to decide upon a finite size. Though we employ scheduling techniques to find the FIFO sizes and maintain the semantics of the model, it is necessary to make the design decisions prudently.

4. MODELING WITH MASIC

In this chapter we apply the MASIC methodology on several examples. We present the grammar based language of MASIC that helps to capture and inject the architectural decisions. The modeling concepts at different levels of abstraction are illustrated by realistic examples, which are implemented using the AMBA on-chip bus architecture. Our results show that the abstract models can be built and simulated much faster than the implementation model at the expense of a reasonable amount of simulation accuracy.

4.1 Introduction

In the previous chapter, we have presented the modeling principles and the design steps in the MASIC methodology. In this chapter, we apply those principles on a number of realistic examples. The system models are implemented using the AMBA on-chip bus architecture, which is often said to be as the de-facto industry standard of on-chip communication architectures.

We begin with the role of communication modeling in the MASIC methodology. Then we discuss different grammar based description styles used within the hardware design community, and the advantage of grammar based descriptions in modeling communication protocols. Next, we present the grammar based language of MASIC that we have developed to describe the system models at different levels of abstraction. This language is tailored towards describing the transaction level models (TLMs) succinctly. We discuss different features of this language, how it helps us to describe the communication protocols of the concurrent processes in an abstract way, and also point out the limitations of this approach. From the grammar based descriptions of the system models, we generate a high-level simulation models in VHDL. Our results show that the abstract TLMs can be built and simulated much faster than a detailed implementation level model.

4.1.1 Communication Modeling in MASIC

A model of a complex system is described as a set of concurrent processes. Concurrent processes executed in a system may be either *independent processes* or *cooperating processes* [124]. A process is said to be independent if it cannot affect or be affected by other processes running in the system. That is, any process that does not share data with any other process in the system is independent. On the other hand, any process that shares data with other processes in the system is a cooperating process. The processes of the dataflow networks, which we have described in section 2.2.3, exchange data among them and belong to the category of cooperating processes.

Concurrent execution of cooperating processes requires mechanisms to allow inter-process communication to exchange data among them, and synchronization among different operations. Buffers provide the general means for the processes to communicate with each other. The size of the buffer and the associated communication discipline to read and write from/to the buffers give rise to different ways of communication, like:

- **Zero capacity**: In this case, the buffer has a maximum length of zero. Therefore, such a link cannot have any token waiting in it. In order for a token transfer to take place, the sender waits until the receiver receives the token. The sender and receiver processes must be synchronized, and communicate in an indivisible atomic action called *rendezvous*. Hoare's *Communicating Sequential Process* (CSP) [73] and Milner's *Calculus of Communicating Systems* (CCS) [103] employ rendezvous communication model.

- **Bounded capacity**: Here, processes communicate through finite size buffers, and uses different synchronization mechanisms. The *synchronous dataflow* (SDF) networks can be schedules with bounded buffer. In the *codesign finite state machine* (CFSM) semantics, developed by Chiodo et al., two machines exchange data using a buffer of size one [31]. The sender emits token and continues running without waiting, whereas the receiver blocks until data is available. The buffer holds the token until the receiver consumes it, or the sender overwrites it with a new token.

- **Unbounded Capacity**: Communication is achieved through unbounded FIFOs. Such a communication scheme implies that the producer process can always write new data in the FIFO (i.e., non-blocking write), but the consumer process may have to wait for the

new data if the FIFO is empty (blocking read). The KPN model [78] belongs to this category.

Communication modeling at different levels of abstraction is the centerpiece of building the *GLObal Control, Communication and Timing* (GLOCCT) of a system. At the functional level, processes communicate explicitly by means of data. There is no control involved, except the presence or the absence of data in the channels.

Process level communication at the *process transaction model* (PTM) level takes place through bounded FIFOs. These FIFOs provide abstract point to point communication channel. Here, an abstract system controller controls the processes such that the producer process must wait if the FIFO is full, and the consumer process must wait if the FIFO is empty. To describe this communication mechanism and account for the architectural limitations[14] of resources, the PTM captures the major synchronization events using a fairly simple control scheme.

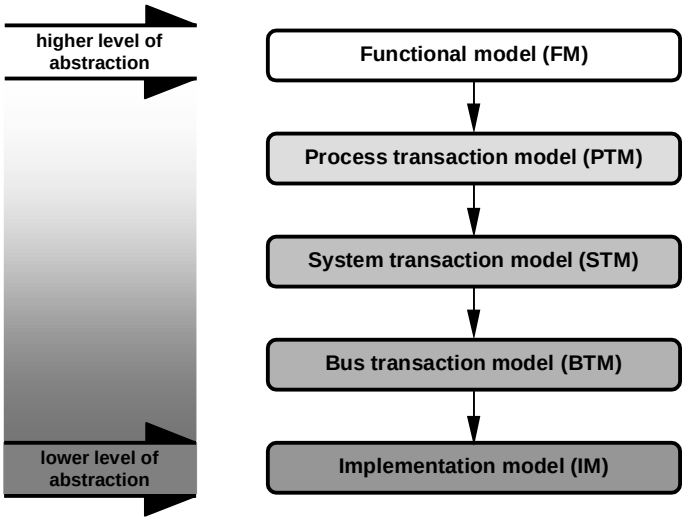

Figure 35. The design flow at a glance

The communication mechanism in STM is not structurally accurate as well. However, it captures the delay due to the communication architecture. The communication at the BTM level is complex. It models communication using the

[14] The synchronization to account for the architectural limitations of the resources are discussed section 3.3.1.

actual pins and wires. Here we describe the structurally accurate component interfaces, like *bus functional model* (BFM) of embedded cores, and the cycle accurate communication protocol, including arbitration logic.

The system design, which we have developed in the previous chapter, is shown in Figure 35. In our design flow, from the functional model through different TLMs to the implementation model, the functionality remains the same. It is only the protocol of data transaction that evolves from the abstract FIFO channels to complex architectures, which involve bus protocols, BFMs, interface adaptation, etc. Therefore, to ease the modeling process, we need an abstract way to describe the protocols.

The protocol defines the language of communication. It describes how two or more processes meaningfully communicate to exchange data. There are different ways of describing a communication protocol. One way is to specify the protocol as a *finite state machine* (FSM) using a *hardware description language* (HDL). However, it is not an abstract way of describing protocols. This is a low level approach where the designer has to describe the protocol using a finite state space, and often needs to deal with the FSMs at the RT level. If the protocol is complex, there can be too many states and the state machine becomes cumbersome. Alternatively, the protocol can be specified algorithmically using nested IF-THEN-ELSE or equivalent structures, which is also hard to understand and debug. Abstract communication modeling has become a part of different system modeling languages like, SystemC [59] and SpecC [48]. However, we argue that grammars provide a more intuitive and natural way of describing protocols than the C++ or C language.

Grammars define the syntax of languages, both natural languages and those used by computers. The first known grammar was the one invented by Panini to describe the structure of the Sanskrit language in ca fourth century BC [19]. In many ways, it had similarities to the grammars used to describe parsers for computer languages.

The success of the grammar based approach can be gauged from the popularity of tools like *yacc*[15] and *bison* in the software domain. They employ a *Backus-Naur form* (BNF) like notation to describe the syntax of typical programming languages. Due to the ease of protocol description offered by grammars, we have developed the grammar based description language to model the communication part of a system model.

---

[15] *yacc* stands from *Yet Another Compiler Compiler*, which is a popular parser used in compiler development. The *bison* is the parser from GNU.

4.1.2 Grammar Based Hardware Specification

Grammars, being the natural way of describing a communication protocol, have been used successfully in protocol specification and HW synthesis. The grammar based *Clairvoyant* system uses grammar rules to describe the control [120]. Their approach is targeted towards detailed specification of communication protocols and interfaces, and control dominated circuits. All interactions with inputs and outputs are described at clock cycle level. Like the approach used in *yacc*, all actions in Clairvoyant are specified in the target language, which is VHDL in their case. In their system, the designer does not describe a particular form of implementation, instead the grammar productions are used by their tool to output an FSM described in VHDL.

Clairvoyant was the progenitor of the *Protocol Compiler* from Synopsys [121]. Protocol Compiler has a graphical user interface to describe the rules in a hierarchical manner along with the associated actions. As a result, design changes at the specification level and debugging become easier. It supports actions both in VHDL and Verilog language. In addition, it has a few built in constructs for specifying simple data manipulation.

Closely related to the grammar based approach, a regular expression based protocol description is proposed in [109]. They address the problem of automatic synthesis of IP interfaces involving incompatible communication protocols. Based on the regular expression protocol description, their tool synthesizes an FSM in Verilog.

The grammar based language *ProGram*—short for *Protocol Grammar*—allows port-size independent specification of communication protocols [140]. Here, port widths are used as constraints to partition the input and output streams. Then the output partitions are scheduled over the available input partitions. The advantage of the port-size independent specification technique is that it allows the designer to explore the port size to trade off area against throughput. ProGram language is targeted towards specifying data communication protocols, and the ProGram compiler generates synthesizable VHDL from the grammar description. The scope of ProGram has been extended to generate hardware software interface and device drivers from ProGram specification [106][107]. To facilitate the high level modeling of signal processing systems, we have developed the grammar based language of MASIC. This language is built upon the ProGram language, with a couple of new constructs and slightly different syntax and semantics.

Recently, Siegmund et al. have presented an extension of the SystemC language using grammar based technique [123]. Their employ their technique to design

communication controller from protocol specification. Two major advantages of their technique are that they allow to design both the partners (that is the producer and the consumer) involved in the protocol from a single specification, and their method is integrated with the emerging system level modeling language – SystemC.

Though the grammar based approaches discussed in this section do not address the problem of system design, they demonstrate two clear advantages, firstly: the ease of protocol specification using grammar, and secondly: the smooth path to HW synthesis from a grammar description. Inspired by these advantages, we have adopted the grammar based description style.

The grammar based ProGram/MASIC languages work in a similar way like a typical parser of a programming language, for example *yacc*, works. Before we start describing the MASIC approach of grammar based description, the next section gives a general introduction to grammars and how they are used in language parsing.

4.2 Grammars for Language Parsing

A parser checks the syntax of computer languages, which is an indispensable part of compiling a computer language. The *syntax* of a programming language describes how the programs would look like, that is how the operators, variables, delimiters, and different programming constructs need to be written. The *semantics* of the language tells us what the program means, that is how we should interpret the meaning of a language construct [10].

A compiler is a program that reads a program written in one language – the *source* language – and translates it into an equivalent program in another language – the *target* language [10]. An important part of this translation from source language to target language is that the compiler to reports to the user the presence of any error in the source program, and generates error/warning messages in an intelligible way. The basic task of a compiler is shown in Figure 36.

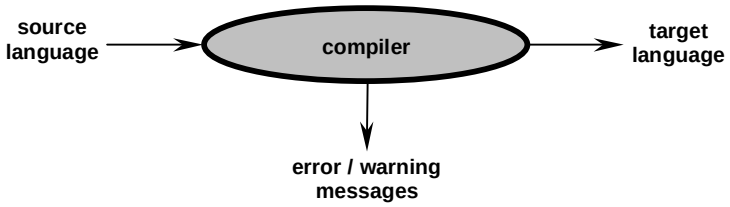

Figure 36. Basic task of any compiler

There are numerous source languages, which range from traditional programming languages like Fortran or Pascal, to specialized *hardware description languages* (HDLs) like VHDL or Verilog. The target language also varies from the machine language of any computer to another programming language. Pattern recognition plays a key role in checking the syntax of this wide range of source programs.

4.2.1 Pattern Recognition

A *pattern* is a set of objects with some recognizable property [11]. There are three different ways of describing a pattern: *finite automaton*, *regular expressions*, and *grammars*.

A finite automaton is a graphical way of representing patterns [11]. Again, there are two types of finite automaton: *deterministic automaton* and *nondeterministic automaton.* An automaton is said to be deterministic if for any state *S* and any input *x*, there exists at most one transition out of the state *S* whose label includes *x*. A deterministic automaton can be easily converted to a program[16] that recognizes patterns. A nondeterministic automaton is, on the other hand, allowed to have two or more transitions out of the same state that have the same input *x*. Nondeterministic automata are not directly implementable by a computer program, but they can be converted to deterministic automata using a method called subset construction [11].

Regular expressions are an algebraic way of describing patterns. UNIX users often use the regular expressions in several commands to search for a file, or to find a

[16] In the case of compiler, this is a software program. In the case of system design, it is an FSM implemented either in software or in hardware.

string of text in a file. Regular expressions can be converted to automata and vice versa.

Grammars, the third way of pattern recognition, use a form of recursive definition to define a pattern. They are more powerful than the regular expressions for describing a language. The proof of this is beyond the scope of this thesis. Interested readers are referred to [11], where it is shown that there are languages for which one can construct a grammar but no equivalent regular expression exists to express the language construct. Grammars provide a *Backus-Naur form* (BNF) like notation to describe the syntax of typical programming languages, such that the hierarchical structure of programming language constructs can be described succinctly.

4.2.2 Language Parsing

The hierarchical analysis of any programming language is called parsing or syntax analysis [10]. The source program is typically read by a lexical analyzer, e.g., *lex*, which produces as output a sequence of tokens. Tokens are representations of strings that simplify the parsing process. Regular expressions are used to specify patterns in *lex*. However, *lex* is not suitable to recognize complex structures, like nested parentheses. The parser performs the complex syntax analysis on the string of tokens, and verifies if the string is acceptable in accordance with the grammar *productions*. One or more production rules make up a grammar of any language. Each line of Figure 37 is a production. In general, a production has three parts [11]:

- *A head*: A syntactic category (i.e., a non-terminal token) on the left side of the colon sign of the figure.
- *The metasymbol* '**:**' : Metasymbols, like '**:**' or '|' do not stand for themselves, instead they mean something else. For instance, the '**:**' means "*can have the form*", whereas the '|' means *OR*.
- *A body*: The right hand side of the colon sign represents the body, which consists of zero or more terminal tokens (i.e., tokens returned by *lex*) and/or syntactic categories.

```
(1)   <digit>  :  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
(2)   <number> :  <digit>
(3)            |  <number><digit>
```

Figure 37. Example of production rules

The rule in the first line of Figure 37 states that a `digit` can be a `0`, or `1`, or … `9`. The next rule spans over line-2 and line-3. The second line says that a single `digit` is a `number`, and the third line says that any `number` followed by another `digit` is also a `number`. As the example shows, the grammar rules can be recursive.

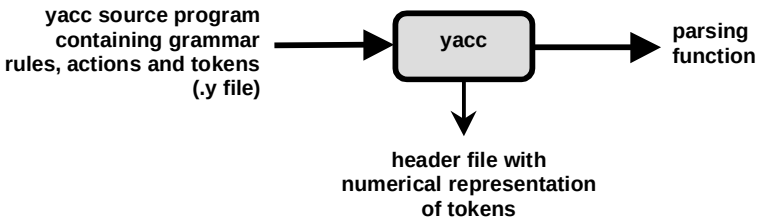

Figure 38. Parser generation with YACC

Yacc is a popular program to convert grammar rules into a parser. In conjunction with the rules, it is customary to specify some *actions*. The actions say what needs to be done, i.e., which construct of the target code needs to be generated, when a particular pattern is matched. Grammar productions and actions are typically written in a file with a `.y` extension. This file also contains the token declarations, which are used by the *lex* program. Figure 38 shows that the *yacc* program reads its input file, and generates both the parser function and the header file with the numerical representation of the declared tokens.

4.2.3 Building a Compiler

Figure 39 illustrates the typical way of generating a compiler. The grammar rules and the token declarations are given as input to *yacc*. It generates the parser function in *yyparse()*, and the numerical representation of the tokens in a header file named *y.tab.h*. The function *yyparse()* typically resides in a file named *y.tab.c*. The *lex* takes the token header file, and reads the lex input file, which contains the token definition in terms of regular expressions. As output, the *lex* program generates the lexical scanner function *yylex()*, which typically reside in a file named *lex.yy.c*.

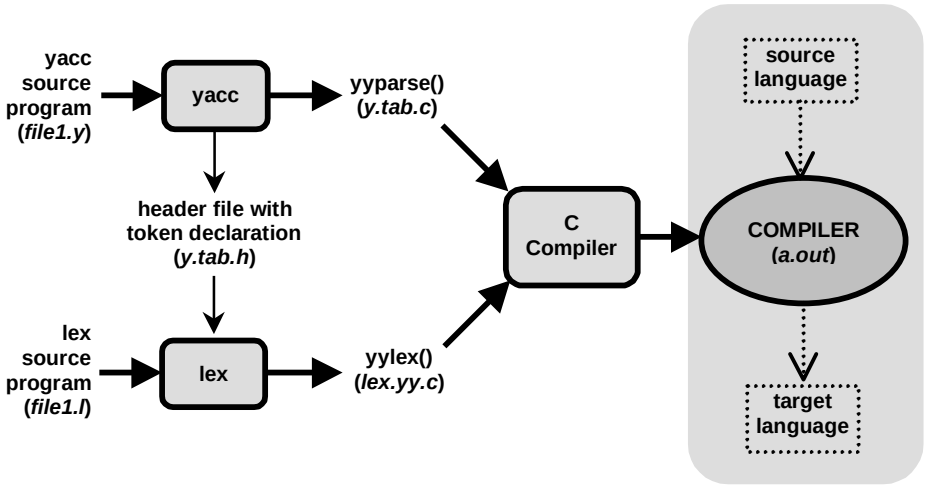

Figure 39. Typical way of building a compiler

A compiler is constructed from these two functions: *yyparse()* and *yylex()*. The *yyparse()* calls the *yylex()* function to scan the program written in the source language. *Lex* matches patterns according to the regular expressions written in the *lex* input file, and returns tokens to the *yyparse()* function. The parser matches the token sequence using the grammar rules written in the *yacc* source file, and generates the program in the target language in accordance with the actions associated to the rules. In the case of ProGram, the source language is a *yacc*-like description of communication protocols, from where it generates an FSM described in VHDL.

4.3 The MASIC Language

Parsers use grammar productions to recognize the patterns of the tokens of a source language, and take a specified action when the particular pattern is matched. In the case of the MASIC language, grammar productions are used to specify a particular signaling pattern. The signaling pattern represents a communication protocol. When a specified signaling pattern is seen at the input, grammar rules generate an output. The outputs are generated using the actions associated with the grammar productions. Typical actions associated to the grammar productions are to generate the synchronization signals; `get` data from the network FIFOs to the buffers in the processes; `put` data from the buffers of the processes to the FIFOs of the network; call a C function with the data saved in a buffer; execute a `wait` statement to model the communication or the computation delays; or simply use a `null` statement. There are two major parts of a MASIC system model description:

- ***Grammar rules***: they specify the invocation of the C functions and the communication protocol that describes the synchronization and movement of data through the architecture corresponding to the level of modeling. Rules can be hierarchically written to achieve compact and readable code.

- ***Constraints to the grammar rules***: they are specified on top of the grammar rules to describe the architectural resources, the signal types, and the interface. They are used to describe buffers, synchronization signals, FIFOs, memories, I/Os, buses, etc. There are four sections of the constraint part: *type definition section*, *import section*, *interface section*, and *storage section*.

```
type definition:                /* define new types */
import section:                 /* import C functions */
interface section:              /* create interface */
storage section:     storage;   /* declare FIFOs, etc. */
grammar rules:       rule;      /* describe grammar rules */
```

Figure 40. Structure of the MASIC description

The overall structure of the MASIC description of a model is shown in Figure 40. The storage and the grammar rule sections start with keywords `storage` and `rule` respectively. This is similar to the *yacc* approach where `"%%"` is used as the delimiter to start the rule section.

Figure 41 shows how a simulation model is generated with MASIC. The constraints together with the grammar rules capture the architectural decisions. The C functions developed during the functional modeling phase are reused to capture the computation of each functional block. The MASIC compiler generates a high level simulation model of the system. The MASIC compiler is built upon the principles of the ProGram compiler [140]. Except a couple of syntactical details, the major improvement of the MASIC over the ProGram is the ability to link to the C functions from the grammar description, such that a DSP function written in C can be fired to perform the computation. The MASIC compiler generates a VHDL model with a link to the foreign language procedure. From the C functions, executables are generated with a `.so` extension. The VHDL simulator looks for an executable with a `.so` extension, makes run-time links to the executable, and performs a cosimulation using the Foreign Language Interface (FLI) of the VHDL.

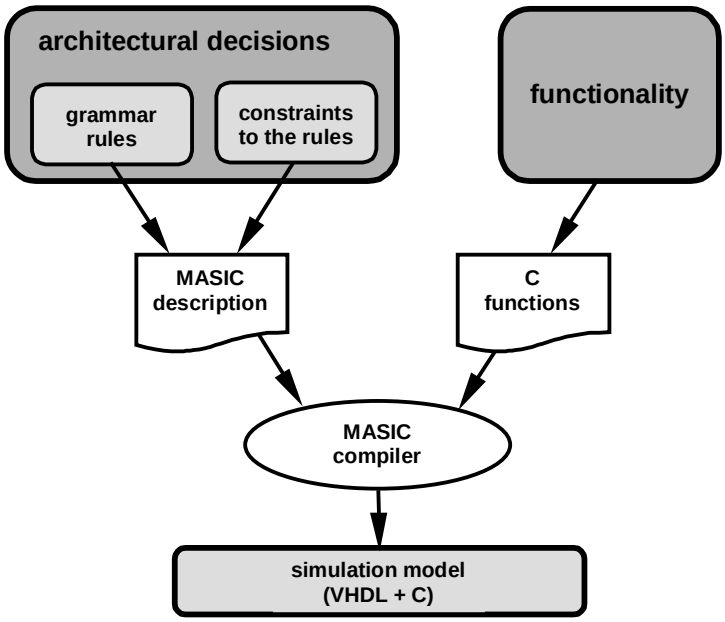

Figure 41. Building a simulation model with MASIC

4.3.1 MASIC Grammar Rules

In the MASIC style of describing a protocol, unlike the *yacc* approach, there is no explicit tokenizer like the *lex*. In fact, tokenization is implied by the sampling of the input stream at the rate of a given clock.

Figure 42 shows the syntax of the MASIC grammar rule that we have devised to describe a communication protocol. We have added an optional `clock_name` on top of the *ProGram*-like grammar rule. Reading different input streams at different speeds symbolizes multiple clocks in a system. Signal processing systems typically include system clock(s), which is a multiple of the sample clock. Grammar rules to describe the process-level communications in abstract models, for example PTM and STM level models, use the sample clock. At BTM level, the communication protocol is described using the system clock. The clock name is optional. The omission of the optional `clock_name` in the rule represents a continuous search for a pattern in the input stream. This is needed to model a combinational behavior, for instance to model a multiplexer or decoder at the BTM level. In the VHDL code generated from the MASIC description, the clock is by default modeled as rising edge triggered.

```
(stream) [@clock_name] : [(condition)]  pattern1 {action1}
                                      | pattern2 {action2}
                                      | .. ..
                                      | .. ..
                                      | reset {action-n}
                                      ;
```

Figure 42. General syntax of a grammar rule

The grammar rule shown in Figure 42 says, a `stream` is read at the rate of the given `clock`. The stream could be a single signal or an aggregate of signals separated by commas. An aggregate stream allows us to describe a protocol that involves multiple signals. For example, to model a situation like, “if the `WRITE` is low and `SEL` is high then generate the `READen` signal” would require to have both `WRITE` and `SEL` as the aggregate and search for a pattern `"01"` in the aggregate stream. Next, the rule says, if a given `condition` is met and a certain `pattern` is found at the input stream, then the associated `action`(s) inside the curly braces must take place. The condition applies for all the branches of the grammar rule.

The action(s) associated with the `reset` is described in VHDL as an asynchronous reset.

The invocation of each function is controlled by a grammar rule. These concurrent grammar rules for the invocation of different functional blocks are arbitrated by the grammar rule of the system controller, which provides scheduling and synchronization. Several grammar rules read their input streams in parallel and have an inherent end recursion. Thus the model represents the concurrent non-terminating behavior of a system.

Let us consider an example to illustrate how the grammar rules are used to control a process. Figure 43 shows a process of a network that reads data from the `in_FIFO` and writes result to the `out_FIFO`. It receives the `data_Rdy` and the `room_Rdy` synchronization signals from the system controller. These signals, respectively, tell the process if there is data available in the `in_FIFO`, and if there is room available in the `out_FIFO`. The process outputs the `read_Rdy` and `result_Rdy` signals to let the controller know that it has finished reading the input data and finished its computation, respectively.

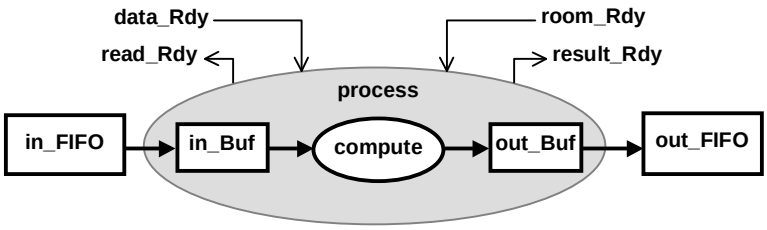

Figure 43. A process of a network

Using the example shown in Figure 24 in section 3.3.1, we have discussed the necessity of synchronization signals to account for the architectural limitations of functional blocks. Here, we assume that the process would be implemented in a resource that has input and output data buffers. Figure 44(a) shows the grammar productions to model this situation. The first rule specifies that if data is ready in the `in_FIFO` then it would be copied to the `in_Buf`, and the `read_Rdy` signal would be asserted so that another process can start writing to this FIFO. In the next line, the `'-'` is used to represent a don't care. This rule says that the `read_Rdy` signal would be deasserted; a C function would execute on the `in_Buf`, and write the result in the `out_Buf`. To model the computation delay, the process waits for an estimated amount of time before asserting the `result_Rdy` signal. The next line

says that when there is room in the `out_FIFO`, the `out_Buf` is copied to the `out_FIFO`; and the `result_Rdy` signal is deasserted.

```
(data_Rdy, room_Rdy) @clk:
      '1', '0'  { get(in_FIFO, in_Buf);
                  read_Rdy  <= '1'; }

      '-', '-'  { read_Rdy  <= '0';
                  call c_fun1(in_Buf, out_Buf);
                  wait for 3 us;   /* computation delay */
                  result_Rdy <= '1'; }

      '-', '1'  { put (out_Buf, out_FIFO);
                  result_Rdy <= '0'; }

    | ...  ...
```

(a)

```
(data_Rdy, room_Rdy) @clk:
      '1', '0'  { null; }

    | '1', '1'  { call c_fun2(in_FIFO, out_FIFO);
                  wait for 5 us;
                  read_Rdy  <= '1';
                  wait until clk = '1';
                  read_Rdy  <= '0'; }

    | ...  ...
```

(b)

Figure 44. Example grammar rules

Figure 44(b) is more interesting. In this case, we assume that the process would be implemented in a resource that neither has an input nor an output data buffer. So, if it receives a `data_Rdy` signal it can not start executing the C function. However, if it receives both the `data_Rdy` and the `room_Rdy` signals, then it executes the C function directly on the data in the `in_FIFO` and saves the result in the `out_FIFO`.

The addition of these synchronization primitives keeps the order of process execution as suggested by a schedule, and adds the necessary synchronization to make the PTM work correctly with different implementation restrictions.

4.3.2 Constraints to the Rules

The grammar productions describe the protocols, whereas the constraints describe the modules of the model onto which the protocol is applied to. In other words, the protocol specifies the language used to communicate over the interfaces and the channels of a system model. Constraints are described on top of the grammar rules to create the system interface to the environment, the interfaces of the components used in the model, the FIFO channels, the synchronization signals, and the C functions that are called from inside the processes. There are four sections of the constraint specification of a model.

```
type spType real[160];
.. .. ..
import C_func ( in_Buf  : in  spType;
                out_Buf : out lpType );
```

(a)

```
package pkg is
  type spType is array(159 downto 0) of real;
  .. .. ..
  procedure C_func (
    variable in_Buf  : in  spType;
    variable out_Buf : out lpType );
  attribute foreign of C_func : function is
                                "C_func C_func.so";
  .. .. ..
end pkg;

package body pkg is
  .. .. ..
  procedure C_func (
    variable in_Buf  : in  spType;
    variable out_Buf : out lpType ) is
  begin
    report "ERROR: foreign subprogram C_func not called";
  end;
  .. .. ..
end pkg;
```

(b)

Figure 45. Importing a C function: (a) using the MASIC grammar constraints, (b) the resulting VHDL code from the MASIC compiler

A. Type Definition Section

Type definitions help to create more compact and readable descriptions. In this section we define types based on the existing VHDL data types. Later on, these definitions are used to declare signals and channels of different widths and array lengths. The keyword `type` is used to define types.

B. Import Section

In this section we declare the C functions that we would like to call from inside the grammar actions. Figure 45(a) shows the MASIC description to define a type and import a C function. The keyword `import` is used to declare a foreign function. Later, using the grammar actions, a process can invoke this function using the keyword `call`. Figure 45(b) shows the resulting VHDL code produced from the MASIC compiler. The type definitions and the import sections are generated in a VHDL package. The VHDL simulator looks for an executable with a `.so` extension, supplies the data in the `in_Buf`, and receives the results in the `out_Buf`.

C. Interface Section

The interface of the system model declares the sources and the sinks of the data streams. It describes the stream name, input/output mode, and the width of the ports. The types defined earlier can be used to define the ports. The clock and reset ports are also defined at the system interface to the environment. Apart from the system interface, we also need to declare the component interfaces to build the BTM level system model.

D. Storage Section

The storage section start with keywords `storage`. In this section we declare different types of internal signals, like the FIFOs, the synchronization signals, etc. These signals can be of user defined types or an existing VHDL type, for example `boolean`.

4.4 Case Studies

We have performed several experiments using three typical signal processing examples: the *linear predictive coding* (LPC), the *ΣΔ demodulator* (SDDM) and the *amplitude modulator* (AM).

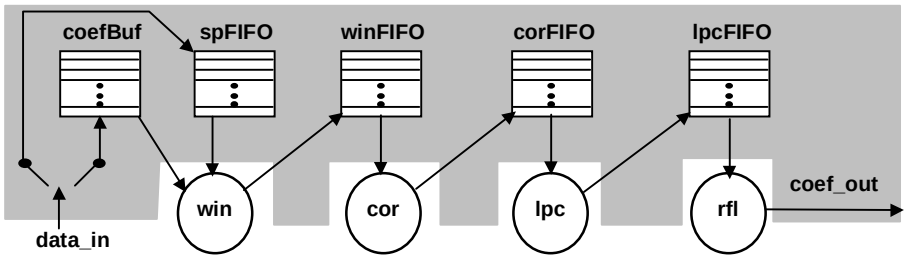

Figure 46. The LPC codec

The speech codec is the main component, which defines the quality of speech in a speech processing system. The LPC codec is basically a frame based analysis of input speech signal. Figure 46 shows the LPC codec. It samples its input values and creates a window of data corresponding to a 20 ms frame. At the sampling rate of 8 kHz, this 20 ms frame yields 160 input samples. Then it performs the hamming window operation on the samples, and generates another 160 data. Next, the autocorrelation function takes this result and computes 11 autocorrelation lags, which the LPC block reads to compute 10 coefficients. Finally, the reflection coefficients are computed from the LPC values.

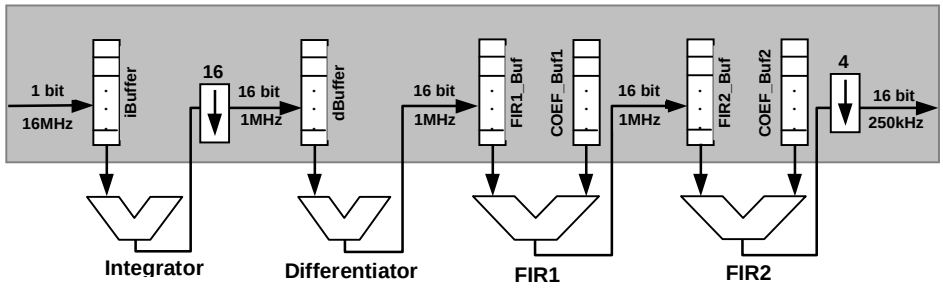

Figure 47. The ΣΔ demodulator

Figure 47 shows the SDDM example. It has one integrator, one differentiator, and two FIR filters. A single bit word from the input is stored in the working buffer of the integrator block at 16 MHz rate. The differentiator receives the decimated output from the integrator and produces results in the `FIR1_Buf`. The FIR1 has a length of 31, and produces output samples at the same rate as its input. The second FIR filter is a decimating one, and it has a length of 69.

The AM example has been presented in [35]. To reduce the number of multiplication operations, it has a specialized structure to realize the amplitude modulation. It consists of two interpolation filters and an add/sub unit.

To illustrate our methodology now we shall go through the whole process with the LPC code example.

4.4.1 Modeling at the FM level

The goal of the functional modeling phase is to validate the functionality at the algorithmic level. At this level, we create the individual C functions. These functions perform atomic execution where reading data from the input FIFO, computing the result from the input sample, and writing the result to the output FIFO is considered as an indivisible operation.

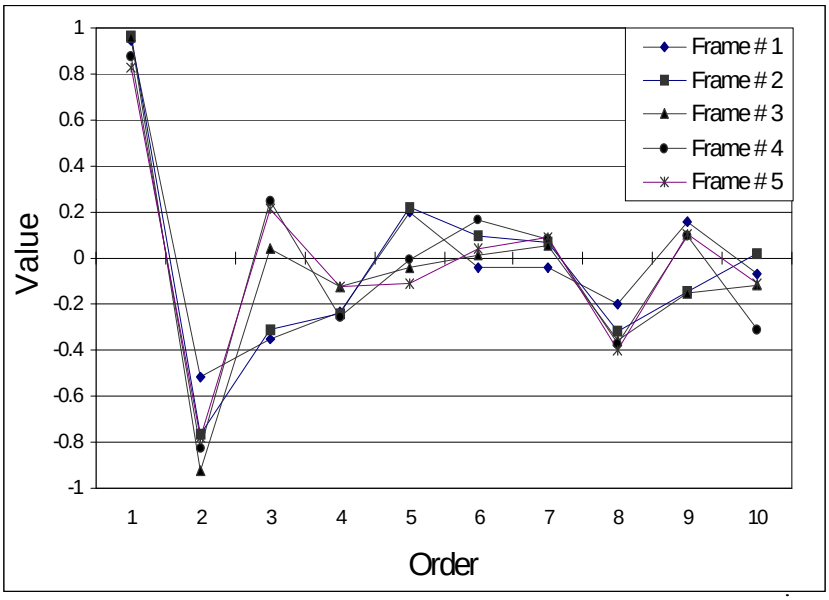

Figure 48. Reflection coefficients computed from five frames of input speech

For the LPC example, the input speech signals were recorded in a SUN workstation using the *soundtool*. All recordings were made in an environment, which is typically referred to in literature as a “quiet office environment”. The

μ-law compressed data has been converted to linear PCM values for acoustic processing of this experiment. Simulation studies are performed with various parameters, for example, algorithm for computing the LPC coefficients, LPC order, etc. The effects of various parameters on the quality of the synthesized speech, and the stability of the speech synthesizer are analyzed. Figure 48 shows the ten reflection coefficients computed from five successive frames of an input speech signal. It shows that the coefficients are bounded between ±1, which will yield a stable filter to synthesize speech using the coefficients.

The outputs of this phase are four C functions that compute the windowing (`win`), the autocorrelation (`cor`), the LPC (`lpc`), and the reflection (`rfl`) coefficients. These functions use real numbers with double precision. At this level, bit-true simulators can be employed to find out the finite word length effects. Table 2 shows the autocorrelation values computed from a frame of input data with and without finite word length effect. For the development of the successive transaction level models we have used the C functions with double precision.

Table 2. Autocorrelation valued with and without finite word length effects

No.	**Autocorrelation values**	
	Without finite word length effect	With finite word length effect (with 20 bits)
1	0.8863	0.8845
2	0.8649	0.8598
3	0.8049	0.8010
4	0.7081	0.7075
5	0.5799	0.5842
6	0.4258	0.4357
7	0.2534	0.2609
8	0.0788	0.0939
9	-0.1015	-0.0822
10	-0.2780	-0.2519
11	-0.4296	-0.4077

4.4.2 Modeling at the PTM level

We reuse the C functions built at the FM level, and add the SLDs using abstract transactions to model the process level communication of the system. At this level we need to decide the computing resources, their schedule, required FIFO sizes, and the major synchronization events. However, no detailed design work is carried out to capture the decisions.

```
/* grammar constraints */

type   spType   real[160];

interface lpc ( input   data_in          real;
                input   data_Available   std_logic;
                input   clk              std_logic;
                reset   std_logic;
                .. .. ..  );

storage;                            /* storage section begins */
array       spFIFO      spType;     /* create FIFO */
internal    frm_Full    boolean;    /* create  synchronization signal */

/* grammar rules */

rule;                               /* rules begin */
(data_Available) @clk :
      '1'   { spFIFO[159..0] <= data_in;   /* load data */
              frm_Full       <= true; }    /* signal status */

    | reset { spFIFO  <= (others => 0.0);  /* initialize with reset */
              frm_Full <= false; }
    ;
```

Figure 49. MASIC description of the system interface and buffering of input data

For the LPC example, we have decided to implement the four DSP functions on four MIPS32-4K processor cores. To find out the schedule we use the synchronous dataflow (SDF) network scheduling technique. We also determine the required FIFO sizes for one iteration over the schedule. We begin with creating the interface of the system to the environment, and then buffering input data in a FIFO. Figure 49 shows the MASIC description for the system interface and buffering of input data. It begins with the `type` definition, which will be used to create the data FIFO. Next, we describe the `interface` of the system. It is worthwhile to note that `reset` is a keyword, which is used to implement exception

in the grammar rules. The keyword `storage` marks the beginning of storage declaration section. In this section, the keywords `array` and `internal` are used to create the data FIFO named `spFIFO`, and the synchronization signal named `frm_Full`, respectively. After the `frm_Full` signal is asserted, the controller may generate the data ready signal to the `win` block.

Next, we describe the grammar rule with a specialized action to accumulate a frame of input data. This grammar rule says that, if `data_Available` is asserted then 160 successive input sample from the `data_in` input is saved in the `spFIFO`. It also says to assert the `frm_Full` signal when the frame is full. The top level of the production always has an implicit tail recursion. Thus, the framing is continued for the stream of input data. The alternative branch of the rule says that, if a `reset` is seen then the `spFIFO` and the `frm_Full` are initialized. The `reset` is implemented as an asynchronous reset. For any other input pattern, for example if the `data_Available` input equals to `'0'`, the rule does not generate any action.

```
/* grammar constraints */

storage;
.. .. ..
array     coefBuf      spType;              /* coefficient buffer */
internal  coef_Loaded  boolean;             /* status signal */

/* grammar rules */

rule;
(data_Available) @clk : (!coef_Loaded)        /* first rule */
      '1'    { coefBuf[159..0] <= data_in;    /* load coefficient */
               coef_Loaded     <= true; }

   | reset   { coefBuf    <= (others => 0.0); /* initialization */
               coef_Loaded <= false; }
 ;


(data_Available) @clk : (coef_Loaded)         /* second rule */
      '1'    { spFIFO[159..0] <= data_in;     /* load data */
               frm_Full       <= true; }

   | reset   { spFIFO    <= (other => 0.0);   /* initialize */
               frm_Full  <= false; }
 ;
```

Figure 50. Modified description to express configurability

Now we add more crucial detail, that is the configuration information, into the GLOCCT of the system. The windowing coefficients are configurable and downloaded at the start-up time using the same input port for data. Further, until all the coefficients are downloaded, the regular processing of data is blocked. In other word, the start-up sequence is reset/initialization, configuration, and data processing. Figure 50 shows the modified MASIC description, which is needed to achieve the configurability. Here, we have added the internal signal `coef_Loaded` to remember the configuration status so that the system does not start data processing before the coefficients are downloaded. The grammar rules are changed as well. The first rule says that if the coefficients are not loaded then the input data will be stored in the `coefBuf`, and at the end the status signal `coef_Loaded` will become true. After the `coef_Loaded` becomes true, regular framing of input data is done by the second rule.

```
rule;
(data_Rdy_win, room_Rdy_win) @clk :
         '1', '0'  { get (spFIFO, in_Buf);
                     read_Rdy_win <= '1'; }
           loop

       | reset     { read_Rdy_win   <= '0';
                     result_Rdy_win <= '0'; }

       ;

loop:  '-', '-'    { read_Rdy_win <= '0';
                     call win(in_Buf, out_Buf);
                     wait for 3 us;
                     result_Rdy_win <= '1'; }

       '-', '1'    { put (out_Buf, winFIFO);
                     result_Rdy_win <= '0'; }

       '1', '-'    { get (spFIFO, in_Buf);
                     read_Rdy_win <= '1'; }

       loop
       ;
```

Figure 51. Grammar rule describing the execution of the windowing block

Now that we have described the buffering of input data and downloading of coefficients, we have the basis for executing the processes of the system. At this level, processes no longer perform atomic execution, instead the process execution

is broken into a sequence of read, compute and write operations. The grammar rules shown in Figure 51 describe the execution of the windowing process. This process receives the `data_Rdy_win` and the `room_Rdy_win` signals from the system controller. This rule is written in a hierarchical way. When it received the `data_Rdy_win` signal it `get` the data and enters the loop. Within the loop, based on the `data_Rdy_win` and the `room_Rdy_win` signals, the process computes the result, `put` the result in `winFIFO` when it gets room ready signal, and again `get` data when it gets data ready signal. For any other combinations of the `data_Rdy_win` and the `room_Rdy_win` signals, except the three specified in the loop, the process does not react. The `reset`, specified in the higher hierarchy is the only way to bring the process out of this loop.

4.4.3 Modeling at the STM level

Communication modeling at the STM level is also abstract, and it is almost like the one at the PTM level. Like the PTM, no physical address is needed at this level. We still have point to point FIFO channels communicating via `get` and `put` procedures. However, the controller behavior is described taking the target communication architecture into account. For example, if the dedicated communication channels between the process at STM level would be implemented using a share channel at the lower level, then to model the actual communication behavior the controller allows only one communication to take place at a time. Again, at the end of each communication the estimated communication time is added in the model to capture the communication delay.

For our case, we have decided to implement the communication architecture using the AMBA AHB on-chip bus [1]. To build the STM from the PTM we add the estimated communication delay. AMBA allows single cycle bus master hand over. The master hand over cycle and the data transfer cycles are counted to estimate the data communication delay. The `wait` statements are used to add the delay in the STM.

4.4.4 Modeling at the BTM level

At the BTM level, the communication is modeled at the cycle accurate level. The communication channels are elaborated according to the AMBA AHB specification [1]. A simplified view of the architecture is shown in Figure 52. The data and control lines are shown in solid and thin arrows, respectively. The

request, grant and wait lines between the arbiter, the bus masters and the slaves are not drawn.

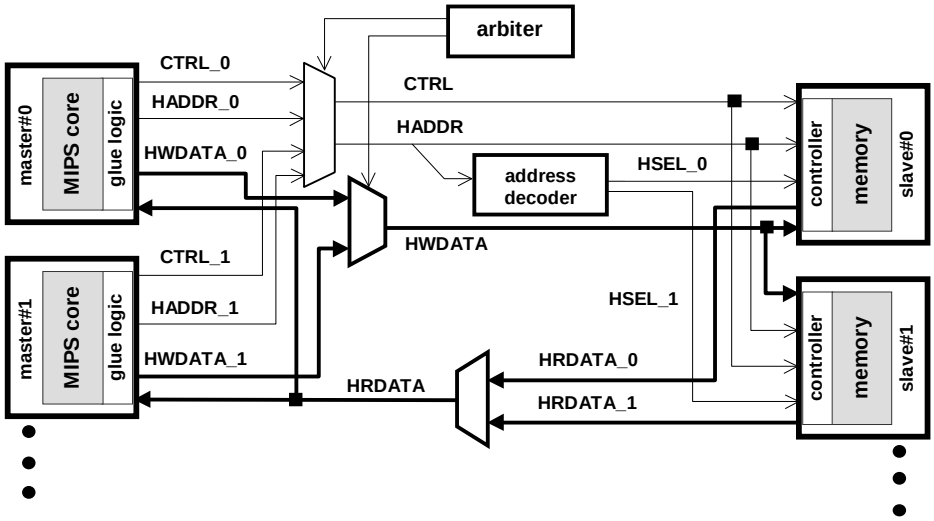

Figure 52. Simplified view of the AMBA architecture

The AMBA AHB uses separate read and write buses operating through a centrally multiplexed architecture. To gain performance, it works in a pipelined fashion where the address-phase of a transfer proceeds simultaneously with the data-phase of the previous transfer. A slave may insert wait states into any transfer, which extends the transfer allowing additional time for completion. For write operations, the bus masters hold the data stable throughout the extended cycles. For read transfers, the salve does not have to provide valid data until the transfer is about to complete. When a data transfer is extended in this way, the address phase of the following transfer has to be extended as well. This is illustrated in Figure 53, which shows three successive single transfers to addresses A, B and C. Besides single transfer, the protocol supports burst transfer of defined length (four, eight and sixteen), and undefined lengths. Figure 54 shows the timing diagram of a burst transfer of undefined length.

At this level we describe the detailed protocol of the communication architecture. We also develop the model of the MIPS32-4K core, which is used as the bus master module, and the glue logic for the interface adaptation between the core and the bus architecture. The core is viewed in two different layers. The outer layer contains the *bus functional model* (BFM) of the core. The BFM represents the external interface of the core, and generates the cycle accurate transactions

supported by the core (e.g., read, write, burst read, burst write, etc.) to the system [6]. The inner layer contains the DSP function in C.

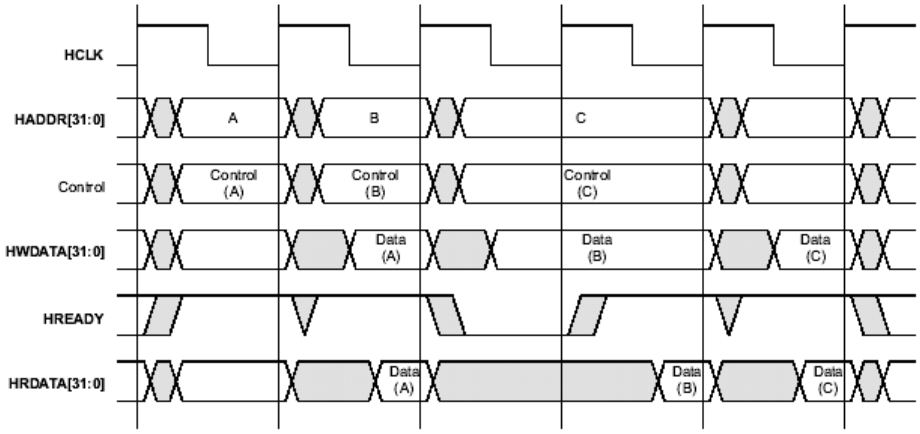

Figure 53. Multiple transfer with wait

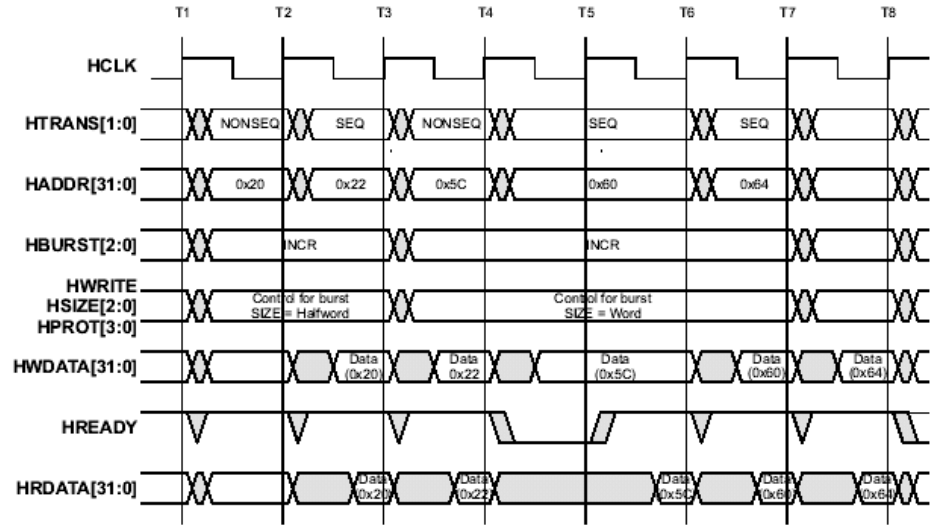

Figure 54. Burst transfer with undefined length

The control, address and data buses are represented as internal signals using grammar constraints. We connect the read and write data buses to the read and write data ports of the bus functional model of the core. The address and control information of the master, which wins the arbitration, are propagated to the slaves through the address bus HADDR. The address decoder, shown in Figure 52, selects a slave by combinational decode of the higher bits of the address bus HADDR. Figure 55 shows the grammar rule for the address decoder. The clock information is absent in the grammar rule, which symbolizes a combinational behavior, and the decoding logic is described using the pattern-action pairs.

```
/* grammar rule for the address decoder */
(HADDR_HIGH_BIT) : "00"  { HSELv <= "0001"; }
                 | "01"  { HSELv <= "0010"; }
                 | "10"  { HSELv <= "0100"; }
                 | "11"  { HSELv <= "1000"; }
                 ;
```

Figure 55. MASIC description of address decoder

Depending on the slave select signal, the appropriate slave unit is selected and the control signal CTRL tells the type of transaction that needs to be performed. The aggregate signal CTRL is composed of several AHB signals like HWRITE, HTRANS, HBURST, etc. Figure 56 shows the grammar description involved in the slave module that includes a memory and a controller. The memory behavior is described as: if the condition CS is true, then a '1' at RWS causes a write and a '0' at RWS causes a read. The next line fetches the address from the address-phase of the transfer. It says to read the address (HADDR) at the arrival of the HCLK if the HSEL signal is high, which is the address-phase. By default, the clock is implemented as rising edge triggered and the reset as an asynchronous reset.

Next, Figure 56 shows that the slave controller reads the aggregate control word, separated by commas. If the first bit pattern is seen, then it asserts the chip select signal and deasserts the RWS so that the RAM outputs (i.e. a read operation) a single word. The second pattern causes a write operation. The third pattern initiates a burst read of unspecified length. The first transfer of a burst uses HTRANS="10", followed by "11" for the remaining transfer and terminates with a "00". This whole information is described as follows: if a pattern of ('1', '0', "10", "001") is seen, the first data of the burst is supplied and then it looks for a pattern labeled as branch1. The branch1, as described below, is repeated while there is a "11" at the HTRANS input; the other inputs are don't cares, represented by the '-'s. Finally, branch1 terminates when a "00" is seen at HTRANS.

```
/* the memory block of slave_0 */
(RWS) : (CS)  '1' { MEM(ADDR) <= HWDATA; }
            | '0' { HRDATA_0  <= MEM(ADDR); }
            ;


/* fetching the HADDR at the rising edge of the HCLK
** of the address cycle */
(HSEL) @HCLK : '1' { ADDR <= HADDR };


/* Protocol of AMBA Slave Controller */
(HSEL, HWRITE, HTRANS, HBURST) @HCLK
      : '1', '0', "10", "000"  { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
      | '1', '1', "10", "000"  { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
      | '1', '0', "10", "001"  { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
         branch1

      | '1', '1', "10", "001"  { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
         branch2

      | reset                  { CS     <= '0';
                                 RWS    <= '0';
                                 HREADY <= '0'; }
      ;

branch1  : '-', '-', "11", '-' { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
            branch1

         | '-', '-', "00", '-'
         ;

branch2  : '-', '-', "11", '-' { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
           branch2

         | '-', '-', "00", '-'
         ;
```

Figure 56. MASIC description of the slave

4.4.5 Modeling at the IM level

The BTM has captured the detailed communication mechanism of the AMBA architecture. The core BFMs represent the structurally accurate interfaces, and they are widely used to model systems before the complete description of the core is available [115]. Since the BFMs perform the cycle accurate transactions (read, write, burst read, burst write, etc.) that the actual cores make, they make sure that the interfacing logic is correct. As a result, the interface adaptation described at the BTM level can be used at the IM level. The BFMs do not include the internal architecture of the processor datapath, and software simulation at the instruction set level is possible with them. To implement the system and simulate the C functions with the architecture now we need to acquire the real cores. Since the real MIPS cores were not available during the time of the experiments, we decided to implement the C functions in hardware. For this purpose, RT level descriptions of the functional blocks were developed.

Two figures of merit have been used to quantify the benefits of this methodology: the simulation time in seconds, and the coding efficiency in terms of word count of the source description. The simulation runs were performed on a Sun Ultra 10 workstation. To avoid the simulation engine start up time, we run it from outside the GUI environment, in a batch mode.

Table 3 shows the VHDL simulation time for the different models of our three examples, the LPC, the ΣΔ, and the AM. The PTM/STM simulates much faster than the BTM, because they do not perform the intricate signaling protocol of data transactions, instead they just use the `get` and `put` procedures to perform bulk transfer. The simulation speedups are highly beneficial in an iterative design flow. The simulation time of the PTM and STM remains more or less the same because these models perform bulk data transfers, and do not differ too much from each other. The IM does not employ the MIPS cores, instead it contain an RTL description of the functions, with a MIPS core interface.

AMBA allows single cycle bus master hand over. The master hand over cycle and the data transfer cycles are considered to make a static estimation of the communication delay. The `wait` statements are used to add the estimated delay in the STM. For the LPC example, we compare the communication time found from the STM with the clock true figures found from the BTM. The communication delay figures found from the STM vary only around ±3% compared to the cycle accurate communication delay figures found from the BTM simulation. The VHDL description of the BTM of the LPC example is synthesized using the Synopsys Design Compiler. The cell area becomes 3784 gates using the *lsi_10k* as

the target technology. This area does not include the cores or the memory cells. Those were used as black boxes during synthesis.

Table 4 shows the code size of the MASIC and VHDL descriptions of the different models for the three examples, and the number of states needed in the VHDL description. The increase in design productivity in term of the design-hour could be guessed from the bulk of VHDL code, and the number of states needed in the VHDL model. In the MASIC approach, the system transactions are expressed in an abstract grammar, from where the VHDL is generated by the grammar compiler.

Table 3. VHDL simulation time

	LPC	**ΣΔ**	**AM**
PTM	13 min 47.7 sec	8 min 36.1 sec	5 min 6.1 sec
STM	13 min 47.7 sec	8 min 36.2 sec	5 min 6.2 sec
BTM	52 min 41.2 sec	37 min 23.5 sec	26 min 54.3 sec
IM	119 min 29.5 sec	78 min 51 sec	46 min 17.4 sec

Table 4. Code size and number of states

	LPC		**ΣΔ**		**AM**	
	MASIC	**VHDL**	**MASIC**	**VHDL**	**MASIC**	**VHDL**
PTM (word count)	305	2596	276	2409	206	1944
STM (word count)	337	2628	300	2447	219	2017
BTM (word count)	2998	15865	2437	11952	1933	9656
IM (word count)	-	18954	-	15208	-	13078
Number of states in BTM	-	176	-	142	-	119

4.5 Limitations of Grammar Based Specifications

The grammar based language of MASIC provides an elegant way of describing the communication protocols. The main advantage is that the grammar based description relieves the designer from expressing the protocols in terms of a finite state space. As a result, the designer can describe the protocol in a natural way in terms of the sequences of the transactions taking place in the system. However the grammar based description style has its own limitations. This is particularly true when the designer already has a state diagram in mind, and wants to describe the protocol of the machine using a grammar.

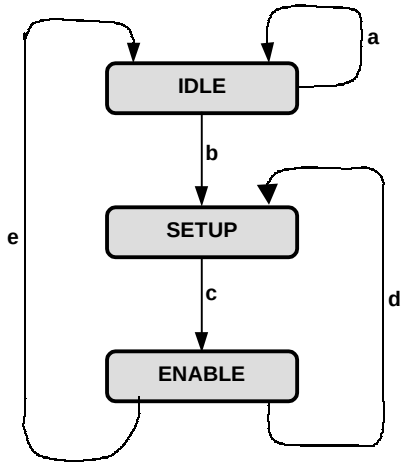

Figure 57. State diagram of AMBA APB bus

Let us take a simple example. Figure 57 shows the state representation of the activities in the AMBA APB bus. Now to describe the activities taking place through all the transitions of this machine, we need to write the grammar rules as shown in Figure 58.

```
(APB) @clk : a  { /* actions */ }
           | b  { /* actions */ }
             branch
           ;

branch : c   { /* actions */ }
         d   { /* actions */ }
         branch
       | c   { /* actions */ }
         e   { /* actions */ }
       ;
```

Figure 58. Grammar rules to describe the example state diagram

Now, if we consider a protocol that can be described using only a few states, and there exist many transitions, which loop back and forth between the states, then describing it in grammar becomes more complex than describing it using a state based technique. However, grammars are better suited to describe protocols where there are long sequences and little branching.

4.6 Conclusion

In this chapter we have presented applied the design methodology on a number of examples. We have presented the grammar based language of MASIC. This language provides an elegant way of describing communication protocol and facilitates us to build system models at different abstraction level. The main advantage of the grammar based description style is that the designer does not need to think in terms of a finite state space, instead he/she can describe the sequences of transactions directly.

Although we have developed the examples using the grammar based language of MASIC, the methodology is not restricted to this particular description language. The steps pertaining to the methodology can be expresses using other languages as well.

We have presented the experimental results of from the three examples that we have considered. Our experimental results show that the higher level models can be designed and simulated much faster than the implementation level models.

Such a speed up effectively helps the designer to explore the design space with different design alternatives.

5. SYSTEM VALIDATION

This chapter presents the system validation approach with the MASIC methodology. First, we discuss the divide and conquer approach to verification of the MASIC model of a system, and discuss its benefits. Next, we present a Petri net based simulation technique of MASIC description of a system model. We describe how a Petri net model is formulated from the MASIC description, and how the Petri net model is simulated.

5.1 Introduction

In chapter 3, we have pointed out that with the increase in the integration of more functionality on a system, the interaction among the functional blocks increases. As a result, the design and validation bottleneck is shifting from the implementation of the individual DSP functions to the *GLObal Control, Configuration, and Timing* (GLOCCT) that composes a system from these DSP functions. The MASIC methodology, besides systematically tackling the design problems, addresses the system validation problem as well.

The verification of complex systems is a difficult problem because many functional blocks placed on a system require a complicated global control for their proper interaction, global resource sharing, and interfacing with the environment. Verifying the individual DSP blocks together with the global control results in a complex validation problem. To tackle this problem, we propose a divide and conquer verification approach to system validation. This approach takes advantage of the systematic separation between the functionality and the architectural decisions of the MASIC model of a system, and develops a technique to facilitate the verification task. Our results show that the divide and conquer verification approach speeds up system verification substantially.

In chapter 4 we have described the MASIC language. A system model developed using the MASIC grammar description can not be simulated unless it is compiled to another language, for instance VHDL. In this chapter we also present a Petri net based simulation technique of the MASIC model of a system. We describe how to formulate a Petri net model from the MASIC description, and how to simulate the Petri net model. As a result of this technique, the need of compiling the MASIC description for the sake of simulation is eliminated.

5.2 The Divide and Conquer Verification Strategy

The divide and conquer verification approach exploits the clean separation between the system functionality and the architectural decisions. The GLOCCT of the system embodies the architectural decisions. To significantly reduce the combined verification complexity stemming from the individual functional blocks and the GLOCCT, we propose that the verification task be split into the following two parts [71]:

- The first part concerns the GLOCCT, and has the objective to verify that *the right amount of known data is available at the right time and place.*
- The second part is concerned with the verification of individual DSP functions, and here the objective is to verify that *if the right data is available at the right time and place, the right result is produced*.

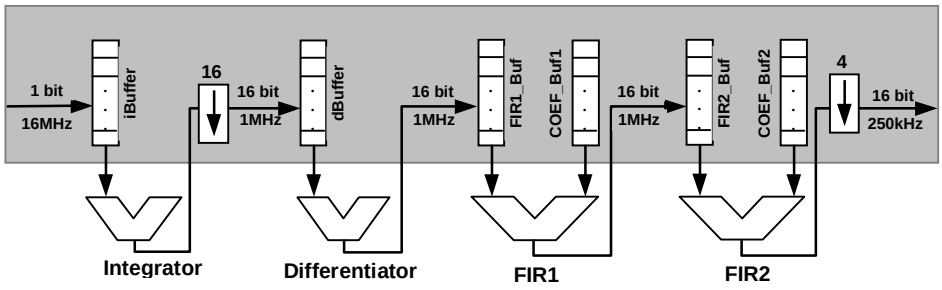

Figure 59. The ΣΔ Demodulator

To justify this claim and define the conditions under which it is valid we will use the ΣΔ demodulator example as shown in Figure 59. It has one integrator, one differentiator, and two FIR filters. A single bit word from the input is stored in the working buffer of the integrator block at 16 MHz rate. The differentiator receives the decimated output from the integrator and produces results in the `FIR1_Buf`. The FIR1 has a length of 31, and produces output samples at the same rate as its input. The second FIR filter is a decimating one, and it has a length of 69. To achieve configurability, we assume that the filter coefficients are downloaded during startup, using the common data input port. We begin by discussing the different aspects of the GLOCCT in the example, and then justify the divide and conquer approach.

5.2.1 GLOCCT and Its Verification

In MASIC, we divide the GLOCCT into the following categories for the purpose of verification. All these instances of GLOCCT are specified in grammar notation, and for the sake of verification, we replace the actual DSP functions by simple mock functions.

A. Start up sequence and configuration

We should differentiate between the start up time interaction and normal operation. Start up sequences often download/initialize parameters for several functional blocks or DSP cores from a few ports that interfaces to some external devices like a CPU. Verifying this implies that the control logic does the initialization in the right sequence. After initialization is over, the affected registers/storage locations have the expected values and in case of error in interaction with the external device, for instance getting an out of range data, the logic can act in the specified manner. Verification of this part of the GLOCCT does not require any interaction with the DSP blocks.

For the ΣΔ example, the startup sequence consists of initializing the data buffers when the power up reset is asserted. Next, coefficients are down loaded from an external CPU, and only then sampling and signal processing is initiated.

B. Interface to the surrounding functionality

In the previous category, we have seen an instance of an interaction with the environment that happens at the start up. Here we are concerned with interaction

with external devices that happens on a more regular basis during the normal operation, and also in cases when an exception arises. However, unlike the previous category of GLOCCT, verifying this category requires pseudo functional block(s) that generates a predetermined pattern of data at the specified rate. There must also be the possibility to inject errors. Such a pseudo functional block is easily fulfilled by a simple counter, which does not strain the simulation engine.

In the ΣΔ example, the interface to the outside world consists of an interface to sample the 1 bit input data stream from the input when the `data_Available` signal is asserted, and producing the 16 bit output data stream at 250 KHz.

C. Internal Resource management

A key category of global control is the functionality that enables sharing of global resources like Buffers, Busses and I/O. As these are direct implications of architectural decisions that are specified in the grammar notation, the global control simulator is able to verify this functionality. As in the previous category, pseudo DSP blocks are required to verify this functionality.

In the SD example, there is no real resource sharing, except for the fact that the same input port is used to download the coefficients of the two FIR filters.

D. Movement of Data

Sampling data and placing them at the right location in the right buffer remain at the heart of any signal processing system. In MASIC, sample buffers are considered as global resources, and as such the sampling process is part of the global control captured in the grammar notation and the GLOCCT is simulated to verify that the samples are stored in the right buffer at right location. The task of verifying that correct arithmetic is performed on the data is left to verification of the individual DSP function, which assumes that correct data is available at the correct location at the correct time. The verification of this category of GLOCCT also requires the pseudo functional blocks.

In the ΣΔ example, each DSP function is associated with a sample buffer, and the GLOCCT consists of sampling the data streams and putting the sampled data at the right place.

5.2.2 Justification

Now that we have elaborated the GLOCCT, and illustrated it with the ΣΔ example, we are in a position to justify the divide and conquer approach that we stated earlier.

First consider the claim that verifying the whole system is equivalent to verifying the GLOCCT and the DSP functions individually. The basis for this is the assumption about no side effects in DSP functions, which means that they monotonically generate data at a certain rate once triggered. That is true for all common signal processing functions. Verifying GLOCCT properties like initialization of buffers after reset, downloading of coefficients and sampling data and storing them at proper places can be done without the presence of the DSP functions. For instance, for the purpose of verifying GLOCCT properties, one can replace the Integrator shown in Figure 59 with a simple counter or look up table that generates known data at the required rate, that is, 16 bits of known data at 16 MHz. If we have such a data stream, we can easily verify the GLOCCT property related to sampling the integrator output, decimation and placing it in the right place for the differentiator. Note that having the known data pattern in the generated data stream simplifies the verification task. For instance, if we have a counter that ticks at 16 MHz, and generates monotonically increasing numbers, we can easily verify the GLOCCT for the differentiator by checking that subsequent numbers deposited in the differentiator buffer differ by 16. Counters or look up tables that replace the DSP functions are called pseudo or mock DSP functions. As these mock DSP functions have the same interface as the real DSP function in terms of operands and timing, plugging the actual DSP functions back would not violate the verified GLOCCT properties.

Verifying other GLOCCT properties such as initialization and downloading of coefficients etc. does not even require the mock DSP functions.

Once the GLOCCT properties are verified, the DSP functions can assume that they have the right data at right place and right time. Based on this assumption, the individual functional blocks can be verified. Whereas known data at the right place at the right time is verified while verifying GLOCCT, right data can be guaranteed because each DSP function is verified individually, where the input data comes from a test bench and is supplied by the designer. Such data is generated from the functional modeling phase.

5.2.3 Benefits of the Divide and Conquer Approach

This splitting of the verification task has several benefits:

A. The complexity of verification is reduced because the designer has to deal with a smaller functionality and a smaller set of properties to be verified.

B. The simulation efficiency is significantly increased since the problem is broken into smaller tasks. The simulation efficiency can be quantified in two ways:

- ***Simulation efficiency measure-1***: This is the more conservative estimation of simulation speedup where the simulation time for the complete system is compared with the sum of simulation time for the GLOCCT and the individual DSP functions for the same number of input samples.

- ***Simulation efficiency measure-2***: Since the divide and conquer strategy splits the system simulation into separate simulations, it allows us to run them in parallel on different machines. With this assumption the second measure of simulation efficiency, which reflects the full potential of the methodology, is the ratio of the complete system simulation time to the maximum of the maximum of the different simulation times taken by any of the simulation involved in the divide and conquer approach.

C. By splitting the verification tasks among GLOCCT and individual DSP functions, one can better utilize the network of workstations that is commonplace in most engineering environments by running individual verification jobs at different machines.

5.2.4 Experiments and Results

To quantify the benefits of the proposed divide and conquer strategy, we have applied it to the ΣΔ example. First, the ΣΔ demodulator is modeled and verified at functional level. This level resulted in four C functions corresponding to the four DSP blocks shown in Figure 59, that is the integrator, differentiator, FIR1 and FIR2. Next, the GLOCCT is described in grammar notation that captures the startup sequence, interfacing with the environment, internal resource management, and movement of data. The MASIC compiler generates a VHDL description of the

GLOCCT, which is simulated with mock DSP functions. These mock functions are simple counters to generate known data. As this is a small example and the results can get skewed by the overheads of the simulation engine, we have simulated the model with 90 million samples to make sure that the simulation time is long enough to ignore the overheads. The C functions for the individual DSP blocks are executed alone maintaining the ratio of input data, for example the integrator is executed using 90 million samples, whereas the differentiator is executed with 90/16 million data samples. To compare the results from the divide and conquer strategy, the complete system is developed in VHDL and simulated for the same amount of data. Table 5 quantifies the benefits of the divide and conquer verification strategy.

Table 5. Quantifying the benefits of the divide and conquer verification strategy

	Simulation time in seconds
GLOCCT simulation with mock functions (VHDL + C)	224
Integrator	95
Differentiator	7
FIR1	34
FIR2	73
Total simulation time for the divide and conquer strategy	433
Complete system simulation (VHDL)	1150
Simulation efficiency measure-1	2.65
Maximum of the simulation time for the simulations involved in the divide and conquer strategy	224
Simulation efficiency measure-2	5.13

5.3 Petri Net Based Simulation

The MASIC model of a system is internally viewed as a set of communicating Finite State Machines (FSM). Here, we present a Petri net based representation of the MASIC model of a system. Unlike the FSM's, the composition of Petri nets does not require a cross product. A composition can be achieved simply by overlapping the output places of the first net with the input places of the second net [111]. Thus the Petri net model does not explode with increasing system complexity. Analysis of the Petri net model shows the clock true transactions of the SOC communication architecture, which includes the synchronization overheads of the blocks. As a result, architectural tradeoffs can be performed and the correctness of the protocol specification can be checked. We begin this section with a brief introduction to Petri net.

5.3.1 Petri Net

A Petri net is a four-tuple [29]:

(P, T, A, w)

where,

P is a finite set of places, $P = \{p_1, p_2, \ldots p_n\}$

T is a finite set of transitions, $T = \{t_1, t_2, \ldots t_m\}$

A is a set of arcs, a subset of the set $(P \times T) \cup (T \times P)$

w is a weight function, w: A $\rightarrow \{1,2,3,\ldots\}$

To model a discrete event system using Petri net, the transitions of a Petri net are used to represent events and places reveal conditions under which events may or may not occur. In a *marked Petri net*, the presence or absence of token(s) in a place indicates whether the condition has been met or not. A marked Petri net is a five-tuple (P, T, A, w, x_0), where x_0 indicates an initial marking. Tokens in the different places of a Petri net show the present state of the net as:

$$\mathbf{x} = [x(p_1), x(p_2), \ldots x(p_n)] \quad (5.1)$$

All the places that have an arc to transition t_j is an *input place* and all the places that have an arc from transition t_j is an *output place* for that transition. The input and output places for transition t_j can be given by:

$$\mathbf{I}(t_j) = \{p_i : (p_i , t_j) \in A\}; \tag{5.2.1}$$

$$\mathbf{O}(t_j) = \{p_i : (t_j , p_i) \in A\} \tag{5.2.2}$$

Now, a transition $t_j \in T$ is enabled if its input places have enough tokens, such that:

$$x(p_i) \geq w(p_i , t_j) \qquad \text{for all } p_i \in \boldsymbol{I}(t_j) \tag{5.3}$$

If a transition is enabled, it can fire. Firing changes the marking of the Petri net and the next state can be given by:

$$x'(p_i) = x(p_i) - w(p_i , t_j) + w(t_j , p_i) \qquad \text{for i = 1, …, n} \tag{5.4}$$

The firing vector **u** is defined as an m-dimensional row vector with all the elements equal to '0' except a single '1' at the j^{th} position, indicating that the j^{th} transition is currently firing. The *incidence matrix* **A** is an (m x n) matrix whose (*j, i*) entry is:

$$a_{ji} = w(t_j , p_i) - w(p_i, t_j) \tag{5.5}$$

The next state $\mathbf{x}'$ of the Petri net is given by:

$$\mathbf{x}' = \mathbf{x} + \mathbf{uA} \tag{5.6}$$

The above equation gives the next state of the system after each firing and is the basis for the analysis of Petri net based simulation of system models.

5.3.2 Building the Petri Net Model

For each FSM we construct a Petri net. Each state of an FSM is represented by a place; we call it a *state-place*. A token in one of these state-places shows the present state of the FSM. Each arc of the FSM is represented by a transition, which model changes of the state.

Signals are modeled in different ways. Signals used for handshaking are called *synchronization signals*, which represent the control signals. On the other hand,

signals used for storage (memory elements) and interconnect (buses) are called *data signals*. The constraint section of a MASIC description is analyzed for these two classes of signals. Each synchronization signal is represented by two places: a *high-place* and a *low-place*. A token in the high or the low-place of a signal respectively represents the high or low state of that signal. A token can not exist in both of the places and both of them can not be simultaneously empty. Data signals are viewed as *placeholders* for data values. They are not a part of the net, instead they are updated with transition firing. Based on the conditions given in the MASIC description, the placeholders are updated either using an assignment operation or using the results of the C functions.

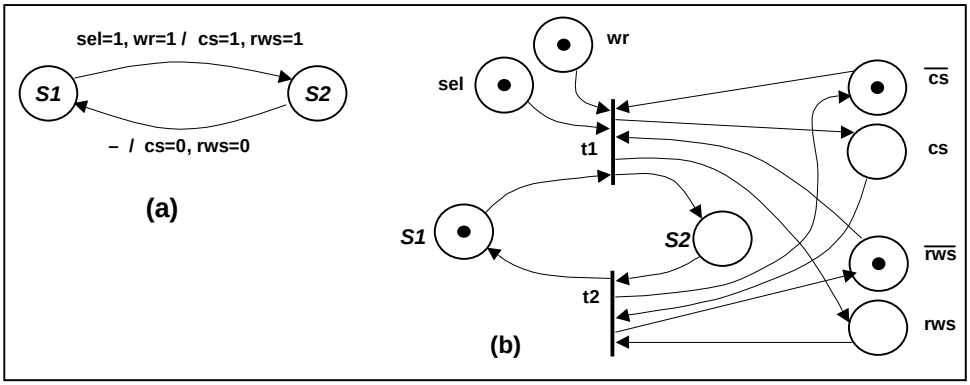

Figure 60. (a) An FSM, (b) The corresponding Petri net model

There is no inherent measure of time in the basic construct of the Petri net. We have introduced a *clock place* to advance simulation time and sequence events with time. The clock signal is represented as a single place, where tokens appear with respect to time and disappears with the subsequent firing. The appearance of a token in a clock-place symbolizes the appearance of a clock edge and advances the simulation time. To model a system with multiple clocks, multiple clock-places are created where tokens are generated according to the ratio of the speeds of the clocks involved. The sequential change of a signal with a clock is modeled as a Petri net transition, which has a clock-place as one of its input places. The combinational change of a signal is modeled by a transition, which does not have a clock-place as one of its input places.

Arcs are drawn between input places to transitions and transitions to output places, which denote the causal relations described in the MASIC specification. The previous state of the FSM (or signal) becomes the input place for the transition

while the next state becomes the output place. If a signal changes the state of an FSM but do not change its own state, then it becomes both the input and output place of the transition.

Besides the synchronization signal and the data signal, we also have *excitation signals*, which are coming to the system from the environment. Figure 60(a) shows an FSM for a section of a memory controller. Here, `sel` and `wr` are excitation signals that come from the environment. The FSM is at state `S1`. Now, if both of these excitation signals are asserted then the machine goes to `S2`, and asserts the `cs` and `rws` synchronization signals, which would cause the data to be written in the memory. The corresponding Petri net is shown in Figure 60(b). The token in `S1` indicates the present state. With the arrival of two tokens in `sel` and `wr` the transition `t1` can fire, which in turn makes both `cs` and `rws` high and causes the data to be written in the memory. In this way, the Petri net model nicely describe the dynamic behavior of the system.

5.3.3 An Example: A Network of FSMs

Figure 61 shows the FSM and the Petri net representation of a portion of a system where an embedded core writes a data value to a memory unit. The system is built using a multiplexed address data bus where simultaneous requests are arbitrated. The core receives an `IO_wr` signal at state `C1` and sends a request to the arbiter to obtain the system bus. After the necessary arbitration process, the core gets the `grant` signal, sends a write request to the memory and places the address on the multiplexed bus. The memory unit sees the request, reads the address and proceeds to state `M2` from `M1`. In the next clock, the core drops the `wrRq` signal, puts the data value on the bus, and proceeds to state `C1`. The memory unit reads the data value as it proceeds to state `M1`.

Each synchronization signal, namely `busRq`, `grant` and `wrRq`, is modeled using two places in the Petri net representation shown in Figure 61(b). The states of the communicating FSMs are copied verbatim to the Petri net model. Arcs have been drawn to show the causal relation between the events. The transition `t2` can fire causing the core to move to state `C2`, and the `busRq` signal to move from low to high. The arbiter diagram is not fully shown. After the arbitration delay, when the arbiter comes to state `A1`, transition `t1` can fire causing the `grant` signal to become high. With the grant signal, the core moves to state `C3`, asserts the `wrRq` signal, and drops the `busRq`. Along with this step, the placeholder for the bus is updated with the address value. Note that the grant signal retains its state during the firing because it is both an input and output place of transition `t3`. With the `wrRq` at high,

the memory proceeds to state `M2`, reads the address value, and updates its placeholder for the address register. In the next clock, both `t4` and `t6` fires, causing the core to drop the `wrRq` signal and put the data on the bus, which the memory reads as it proceeds to state `M1` from `M2`.

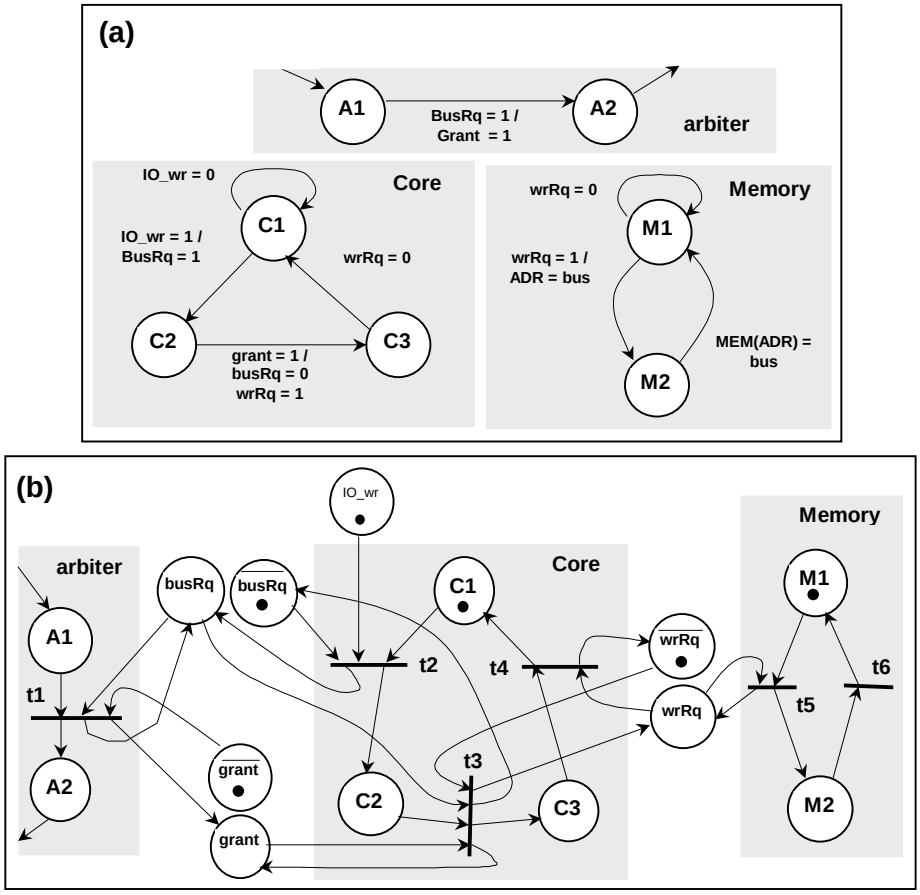

Figure 61. An Example: (a) Communicating FSM, (b) Corresponding Petri net representation

5.3.4 Simulating and Analyzing the Net

Figure 62 shows the overall flow of the Petri net based simulation. The FSM representation of the MASIC model is converted into a Petri net representation. All transitions in the example shown in Figure 61(b) have the clock-place as one

of their input places. Hence none of these transitions can fire before the next clock comes. Sufficient tokens are placed in the clock place so that all enabled transitions at a given time can fire. As shown in the previous example, transitions `t4` and `t6` fire simultaneously without causing any non-determinism. Non-determinism occurs if the firing of one transition disables another enabled transition. FSMs are supposed to work deterministically. If non-determinism exists in the net that reveals an erroneous specification.

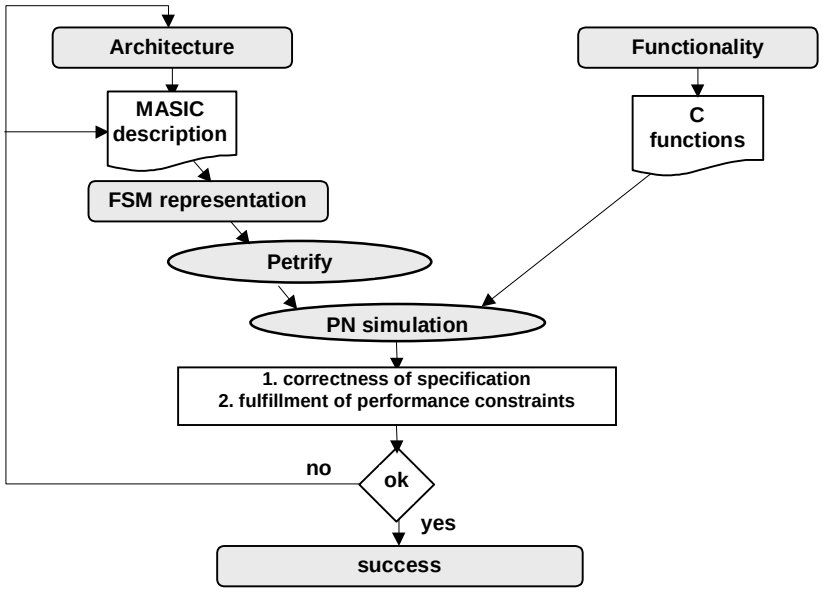

Figure 62. Petri net based simulation flow

We have used terms like high-place or low-place to indicate what a token in that place means, and we have used clock-place to indicate the arrival of a clock. There is no distinction among these places from a Petri net point of view. Thus the incidence matrix for the net can be deduced using the equation-5.5. The simulation procedure begins with the formulation of the incidence matrix. The size of the matrix will depend on the size of the Petri net. The size of the Petri net does not explode because all the states of the set of communicating FSMs are copied verbatim and all the synchronization signals are represented using two places in the Petri net. The total number of places in the Petri net is bounded and this number would be equal to the summation of all the states of the communicating FSMs plus twice the number of the synchronization signals, because each synchronization signal is represented by a high-low pair. The net is simulated using the procedure shown in Figure 63.

1. Incidence matrix is formed for the PN derived from the MASIC description of the system
2. Reset signal gives the initial states of the signals and the communicating FSMs
3. Clock place gets tokens to advance simulation time
4. Using the present state of the net, enabled transitions are marked, that gives the firing vector
5. Enabled transition(s) can be fired at any order
6. Next state of the system is computed using equation 6
7. Based on the new state, if a given condition is true then update the placeholders for data values using an assignment operation or a C function call
8. Repeat from step 3 as long as there are enabled transitions

Figure 63. Simulation procedure

With each firing, the simulation time advances and the Petri net model gives the new state of all the FSMs and the signals. The placeholders for data elements also get updated. The C-functions are invoked on these data value to compute the result. The firing procedure provides the synchronization of the data flow to the placeholders and the execution of the C-functions on the data value.

5.3.5 Experiments and Results

We have applied the Petri net based analysis technique on the Linear Predictive Coding (LPC) system shown in Figure 64, which computes the LPC coefficients. The system samples input speech signal at a rate of 8 kHz, buffers 160 samples, which corresponds to 20 ms frame of input data. Then it performs the Hamming windowing (HAM) operation on the buffered speech. Next the autocorrelation block (COR) reads the 160 samples stored by the Hamming block and computes the autocorrelation values. Finally, the LPC block takes these values, computes the LPC coefficients and outputs them.

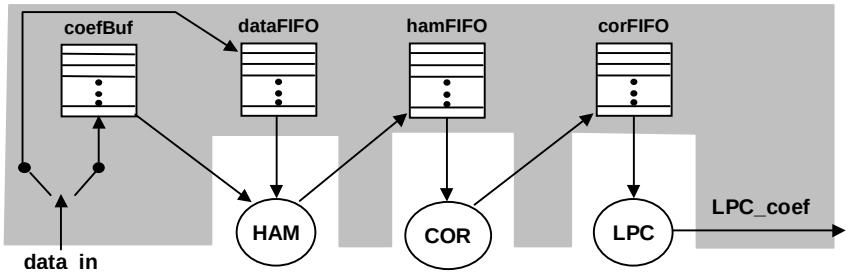

Figure 64. Computing LPC coefficients

We add a few more details regarding the startup sequence of the system, like the windowing coefficients are downloaded using the same data input port. The regular processing of data begins after downloading is finished and valid data is present at the input. We describe the process transaction level model (PTM) of the system. These handshaking signals of the PTM are viewed as synchronization signals in the Petri net model. On the other hand, the signals from the environment, like the "valid data present at input" is viewed as an excitation signal. These signals and the FSMs for each of these blocks constitute the Petri net model of the system. The reset signal gives the initial distribution of tokens in the places from where the firing vector is computed.

The arrival of the sample clock fires the enabled transitions, and gives the next state of the model, the new states of the synchronization signals, and advances the simulation time. Corresponding to the time, the input data value is given in a file from where the placeholders of the Petri net model are assigned with the data. After the coefficients are loaded and the `dataFIFO` is full, the `HAM` block is invoked and the `hamFIFO` is updated using the windowing function. The subsequent processing is done in a similar fashion. System simulation at this level shows that the startup sequence and the inter-process synchronization are taking place correctly.

Next, we elaborate the data transactions to the bus transaction level. We decide to perform the DSP functions on three different cores. Hence the abstract function interfaces of the PTM is elaborated to the BFM of the core. The abstract channels are reined to the bus protocol of the target implementation. The buffers are implemented as a centralized shared memory. Here we have built the bus architecture in two different ways. In the first case, we connect the cores via a multiplexed address-data bus. Later, we change the system specification to model a separate bus for the address and data. Since the functionality remains unchanged, the same C functions used in the PTM level are used at the bus transaction level. A

Petri net model for each of the descriptions is developed. We simulate both of the models with one frame of input data. Figure 9 shows the number of cycles needed to transfer data using these two different bus architectures.

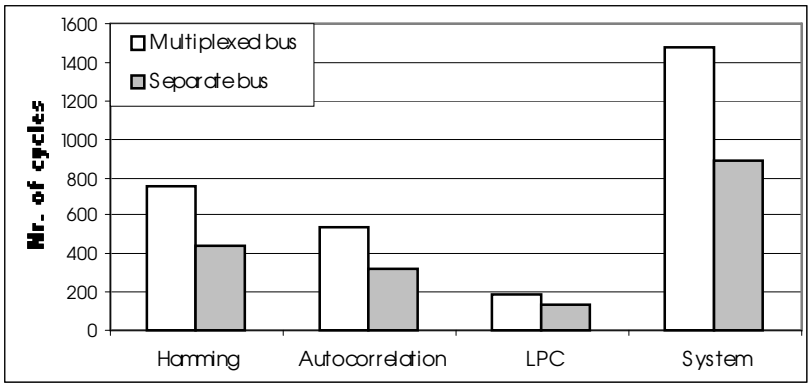

Figure 65. Data transaction through multiplexed and separate address and data bus

Table 6 shows the simulation time taken by the bus transaction level Petri net model and the RT level implementation of the VHDL model for 1 sec of input data. The simulation runs were performed on a Sun Ultra 10 workstation. The VHDL simulator was not run from a GUI environment, instead a batch mode simulation was performed from outside the GUI. The Petri net model uses C functions to perform the computational part of the model, and runs much faster compared to the RT level VHDL model.

Table 6. Simulation time

	Simulation time	
	Multiplexed bus	**Separate bus**
Petri net model	53 sec	31 sec
VHDL model	8 min 36 sec	5 min 14 sec

The Petri net representation of the system model paves the way to verify the system model using different Petri net based analysis techniques. We use the conservation property of the Petri net model to perform specification verification to check the correctness of the input specification.

Conservation refers to the property of a Petri net to maintain a fixed number of tokens at each state. In our model, one of the state places for an FSM must have a token, which indicates the present state of that FSM. Again, a token must exist in one of the places (high-place or low-place) of a synchronization signal. Both of the places can not be empty at the same time. Hence the Petri net model is checked for the conservation property. The process transaction model first failed to hold the conservation property. None of the places for the "coefficient loaded" signal had a token at the beginning. This was due to the missing statements associated with the reset signal. The problem was corrected by incorporating the initialization statement in the action section of the grammar description of the reset signal. While this situation shows a conservation violation at the initial state, the property could be verified during the dynamic operation of the net, using coverability tree analysis.

5.4 Conclusion

We have presented a divide and conquer strategy to address the verification bottleneck of a system model. This strategy relies on splitting the verification task in two activities. The first activity is concerned with verifying the Global Control, Configuration and Timing (GLOCCT) and the second activity is concerned with verifying the individual DSP functions. Justifications and benefits of the divide and conquer strategy have been discussed. The experimental results show that substantial simulation speedup can be achieved using the divide and conquer approach to system verification.

We have also presented a Petri net representation of the system model, which makes it possible to simulate a MASIC model without having to compile it to another language.

The MASIC model of a system is internally viewed as a set of communicating Finite State Machines (FSM). Though FSMs are good for modeling sequential behavior, modeling concurrency and memory is difficult due to the state explosion problem. A single machine mimicking the concurrent behavior of a set of machines would need a number of states equal to the product of number of states of each machine. A memory would need as many states as the number of values

that can be stored at each location raised to the power of number of locations. The number of states alone is not always a good indication of complexity but has a strong correlation [50]. On the other hand, the composition of Petri nets does not require a cross product. A composition can be achieved simply by overlapping the output places of the first net with the input places of the second net [111]. Thus the Petri net model does not explode with increasing system complexity. As a result, the Petri net representation would be of great value to verify the system using established Petri net based technique. For example, the bounedness property of the net can be checked. Boundedness refers to the property of a place to have a certain number of tokens. In our Petri net model, there cannot be more than one token in a place for FSM state or synchronization signal, at any instance of time. If token starts to grow in a state place that would indicate that there exists more than one next state of present state for certain input. This reveals nondeterminism in the FSM, which is not acceptable. Again, more than one token in a place for a synchronization signal reveals that more than one FSM is writing to a single signal. The boundedness property can be shown to hold using the coverability tree. In addition, a deadlock situation between two cores contending for a shared resource could be found out from the reachability analysis of the Petri net. It would be of great advantage to exploit these Petri net based analysis techniques effectively to validate the MASIC model of a system.

6. CONCLUSION

This chapter summarizes the current work, and discusses its benefits. Finally, the limitations of the work, and the directions to future improvements are presented.

6.1 Summary and Conclusion

We have presented the MASIC methodology for signal processing systems design. The salient features of the methodology are the clean separation between the functionality and the architectural decisions, the systematic categorization of the architectural decisions, and the stepwise capturing of the architectural decisions through a number of transaction level models. The sequence of steps through which the methodology yields an implementation model of a system are described elaborately.

In MASIC, we begin with a functional model of the system. The functionality is the abstract behavior of the system that tells us what the system is supposed to do, and it is free from any implementation details. Next, the architectural decisions are captured to map the functionality on the system architecture. These decisions tell us how the functionality is implemented, that is how the individual functional blocks perform their computation, and how these blocks communicate with each other and with the environment.

The separation between the functionality and the architectural decisions are of great value for the design methodology. As a result of this separation of design concerns, the design complexity is split into two orthogonal axes, namely communication and computation. The architectural decisions, regarding the implementation of the GLObal Control, Configuration, and Timing (GLOCCT) enters in the design flow much earlier than the detailed RT level design. Hence, it

becomes easier for the designer to explore the design space at a higher level of abstraction, where less amount of details are needed to be dealt with.

There are a huge number of architectural alternatives to implement both the computation and the communication aspects of a system. Exploring all the architectural decisions is an inefficient approach. We have presented a systematic approach to categorize the architectural decisions such that the designer does not need to consider all the possible alternatives at once. We classify the architectural decisions in two categories: *system level decisions* (SLDs) and *implementation level decisions* (ILDs). As a result of this categorization, we only need to consider a subset of the decisions at once. We add the SLDs to the functional model to create an abstract intermediate model. Next, the ILDs are added to the intermediate model to build a more detailed model. The categorization of the design decisions allows us to create intermediate models at different levels of abstraction, which in turn eliminates the top-down iteration present in more conventional design flows.

Mapping the functionality on a heterogeneous architecture is a nontrivial task. If the initial design decisions do not yield satisfactory performance, then changing the decisions require a major redesign effort when they are captured with greater detail at a lower level of abstraction. To minimize the expensive redesign effort we define three *transaction level models* (TLMs), which reside at different levels of abstraction between the functional and the implementation model of a DSP system. These TLMs capture the design decisions using abstract transactions where timing is modeled only to describe the major synchronization events. As a result, the functionality can be mapped to the system architecture without meticulous details. Also, the artifacts of the design decisions in terms of delay can be simulated quickly. Thus the MASIC approach saves both modeling and simulation time.

It is of course possible to go directly to the IM level from the FM level. In this case, the SLDs and the ILDs need to be described elaborately in a single step. This clearly involves a huge design effort, and incurs a long design time. If, at the end, the detailed model does not meet the performance goals then an expensive redesign effort becomes inevitable. The purpose of creating the intermediate TLMs is to formally capture different aspects of a system in a systematic way. As a result the long design effort can be broken into short incremental design steps, the intermediate models can be checked against performance requirements at different abstraction levels, and the design decisions can be changed if needed.

However, the creation of a number of TLMs may tend to be not so desirable in certain cases. In such a case a designer can combine some of the steps. As a result,

it becomes possible to skip the creation of an intermediate TLM, and move to the next level of abstraction. However, the redesign effort to change the design decisions to meet performance would be higher in such a case.

To cope with the time-to-market pressure, reuse of the existing hardware/software components is a crying need at all levels of abstraction. The MASIC methodology effectively reuse the C functions developed at the functional level to build the abstract system models, like the process transaction model (PTM), the system transaction model (STM), and the bus transaction model (BTM). Again, it helps to reuse the predesigned and preverified hardware blocks using their interface descriptions. The BTM level model contains the structurally accurate interfaces of hardware blocks, or the bus functional models (BFMs) of processor cores.

Another point to note here is that the methodology effectively complements the popular modeling style of signal processing systems. Starting from the existing and popular modeling formalisms like the Kahn process network (KPN), or the synchronous dataflow network (SDF), we describe the design steps to realize a model. As a result, the acquaintance period for a typical designer would be very short.

To successfully exploit the advantages offered by the MASIC methodology there is a need to have adequate tool support to build and simulate the models. In our design flow, from the functional model through different TLMs to the final implementation, the functionality remains the same. It is only the protocol of data transaction that evolves from abstract FIFO channels to bus protocols and component interfaces. To build the system models at different abstraction levels, we have presented the grammar based language of MASIC. The grammar based language relieves the designer from expressing the protocols using a finite state space, and instead allows describing the protocol in a natural way in terms of the sequences of the transactions.

It is not possible to simulate the grammar descriptions of the system models, unless they are compiled to another language, for example VHDL. To get around this and to make it possible to simulate the grammar descriptions, we have presented a Petri net representation of the models. As a result, the MASIC models can be simulated without having to compile them to another language.

Verification of complex systems is a difficult problem because many DSP blocks placed on a system require a complicated global control logic for their proper interaction. Verifying the DSP blocks together with the global control results in a complex validation problem. To address the verification problem, we have described a divide and conquer approach. This approach takes advantage of the

clean separation between the functionality and the architectural decisions in the MASIC model of a system, and speeds up the verification process substantially.

In summary, MASIC offers a smooth path from the functional modeling phase to the implementation level, describes design steps to capture the design decisions early in the design flow, facilitates the reuse of HW and SW components, and enjoys existing tool support at the backend.

6.2 Limitations and Future Work

In order to build CAD tools to work on a system model, it is necessary that the tool "understands" the semantics of the model. The intermediate TLMs formally capture different aspects of a system in a systematic way, and there are clearly defined semantics of these TLMs. This paves the way to build design tools to automate the design steps through the TLMs. However, currently the design steps from one abstract level to the next are carried out manually. The development of the automation tools would significantly ease the design flow from the functional level through the TLMs to the implementation level. It is, however, important that the automation tools provide the user with enough interaction and controllability.

Currently we start from a schedule of atomic process execution, split the execution into a sequence of read, compute, and write operations, and manually determine their order to properly synchronize their interactions. It would be of great value to devise an automated technique to perform this task. Moreover, we assume that processes communicate only at the beginning and at the end of the computation. However, there are situations when it is needed to model that the communication and the computation are in fact interleaved. The problem of scheduling task graphs with interleaved communication and computation has been addressed in [32].

Although the MASIC methodology is not restricted to synchronous dataflow (SDF), all the examples that we have considered here belong to the SDF category. There are signal processing applications, that has data dependent behavior with soft real time constraints. To model a general process network we need to develop a dynamic scheduler. In this case the number of tokens communicated through the channels can not be known in advance. As a result, determining the communication delay would become difficult, because data communication conflicts can not be known in advance. To address this problem, Kumar et al. presents an estimation technique of possible data communication conflicts during mapping the process networks to multiprocessor architectures [47].

Currently the interfacing adapters are written manually. However, this task can be automated using the ideas presented in [109]. Currently the communication delay is not estimated using any rigorous technique. The simulation accuracy of the STM can be improved by using the delay figures found from the trace based analysis technique shown in [88].

The methodology, as it stands now, only considers the consequence of a design decision in terms of computation and communication delay. Though delay is an important factor for the kind of systems that we are interested in, the power consumption is also a significant issue. In the future it is needed to incorporate the power estimates in the TLMs so that the design decisions can be explored in accordance with power/energy constraints.

To map the functionality on a system architecture, we have relied on design decisions. Another way of moving to an implementation level is to employ semantic preserving transformations [117]. Semantic preserving transformations do not change the meaning of the model, and are mainly used for design optimization during synthesis. Design decisions, on the other hand, may change the behavior of the model. For example, the PTM with a finite sized FIFO may not behave in the same way as the functional model does when an artificial deadlock occurs, which would not have happened if the FIFOs were infinite. Since infinite FIFOs are not practically realizable we need to decide upon a finite size. Though we employ scheduling techniques to find the FIFO sizes and maintain the semantics of the model, it is necessary to make the design decisions prudently.

The grammar based language of MASIC provides an elegant way of describing the communication protocols. This is because the grammar based description relieves the designer from expressing the protocols in terms of a finite state space. As a result, the designer can describe the protocol in a natural way in terms of the sequences of the transactions. However the grammar based description style has its own limitations. Describing a protocol that does not involve too many states but includes a large number of transitions, which loop back and forth, is complicated in grammar than in a state based description. This is particularly true when the designer already has a state diagram in mind, and wants to describe the protocol of the machine using a grammar.

The examples presented in this dissertation are described using the grammar based language of MASIC. However, the steps pertaining to the methodology is not restricted to the grammar based description style. The TLMs can be described using other system level languages, like SystemC or SpecC. In the future, it would be nice to apply the modeling style of MASIC using languages like SystemC or SpecC, and compare the modeling efforts.

The divide and conquer verification is simulation based. In the future it would be interesting to explore the possibility of employing model checking techniques to prove the GLOCCT properties in a formal way. The Petri net representation is currently used only to simulate system models. However, the Petri net representation also paves the way to apply different Petri net based analysis techniques, like deadlock or reachability analyses, which have not been explored.

Finally, the MASIC methodology has been developed targeting DSP applications. For DSP applications, the functionality is data dominated, and the C functions nicely express the computational part of the model. On the other hand, the implementation architecture is control and communication intensive, and the grammars nicely describe them. However, if we consider different application areas, like industrial control, etc., then the functionality itself becomes control oriented. Then it would make more sense to express both the functionality and the architecture in grammars, in such a way that the functional and the implementation aspects are still separated from each other.

7. REFERENCES

[1] AMBA on-chip bus specification [Online]. Available: http://www.arm.com

[2] Cadence: SPW, [Online]. Available: http://www.cadence.com

[3] International Technology Roadmap for Semiconductors (ITRS), Design: 2001 Edition, [Online]. Available: http://public.itrs.net

[4] International Technology Roadmap for Semiconductors (ITRS), Design: 2003 Edition, [Online]. Available: http://public.itrs.net

[5] Mentor Graphics: DSP Station, [Online]. Available: http://www.mentor.com

[6] MIPS32 4K Processor Core Family Integrator's Manual [Online]. Available: http://www.mips.com

[7] Virtual Component Exchange (VCX) [Online]. Available: http://www.thevcx.com

[8] Virtual Socket Interface (VSI) Alliance [Online]. Available: http://www.vsi.org

[9] S. Abdi, D. Shin and D.D. Gajski, "Automatic communication refinement for system level design," in *Proc. Design Automation Conf. (DAC)*, pp. 300-305, Jun. 2003.

[10] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[11] A. V. Aho and J.D. Ullman, *Fundamentals of Computer Science*, Computer Science Press, W.H. Freeman and Company, New York, NY, 1995.

[12] A. Allan, D. Edenfeld, W.H. Joyner Jr., A.B. Kahng, M. Rodgers and Y. Zorian, "2001 Technology roadmap for semiconductors," *IEEE Computer*, vol. 35, no. 1, pp. 42-53, Jan. 2002.

[13] S. Bakshi and D.D. Gajski, "Partitioning and pipelining for performance-constrained hardware/software systems," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 4, pp. 419-432, Dec. 1999.

[14] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, pp. 45-52, Apr. 2003.

[15] E. Barros, W. Rosenstiel and X. Xiong, "A method for partitioning UNITY language in hardware and software," in *Proc. European Design Automation Conf. (Euro-DAC)*, pp. 220-225, 1994.

[16] T. Basten and J. Hoogerbrugge, “Efficient execution of process networks,” in *Proc. Communicating Process Architectures*, pp. 1-14, IOS Press, Amsterdam, 2001.

[17] L. Benini, A. Bogliodo and G. de Micheli, “A survey of design techniques for system-level dynamic power management,” *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 8. No. 3, pp. 299-316, Jun. 2000.

[18] G. Berry, “The foundations of Esterel,” in *Proof, Language and Interaction: Essays in Honor of Robin Milner*, G. Plotkin, C. Stirling and M. Tofte, Ed. MIT Press, 1998.

[19] A. Bharati, V. Chaitanya and R. Sangal, *Natural Language Processing – A Paninian Perspective*, Prentice Hall India Ltd., 1995.

[20] G. Bilsen, M. Engels, R Lauwereins and J. Peperstraete, “Cyclo-static dataflow,” *IEEE Trans. Signal Processing*, vol. 44, no. 2, pp. 397-408, Feb. 1996.

[21] P. Bjuréus and A. Jantsch, “Modeling mixed control and dataflow systems in MASCOT,” *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 5, pp. 690-703, Oct. 2001.

[22] P. Bjuréus, *High-level modeling and evaluation of embedded real time systems*, Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2002.

[23] I. Bolsen, H.J. De Man, B. Lin, K.V. Rompaey, S. Vercauteren and D. Verkest, “Hardware/software co-design of telecommunication systems,” *Proc. IEEE*, vol. 85, no. 3, pp. 391-418, Mar. 1997.

[24] J.Y. Brunel, E.A. de Kock, W.M. Kruijtzer, H.J.H.N Kenter, W.J.M. Smits, “Communication refinement in video systems on chip,” in *Proc. 17th Int. Workshop on Hardware/Software Codesign (CODES)*, pp. 142-146, May 1999.

[25] J.Y. Brunel, W.M. Kruijtzer, H.J.H.N Kenter, F. Petrot and L. Pasquier, “COSY communication IP’s,” in *Proc. Design Automation Conf. (DAC)*, pp. 406-409, Jun. 2000.

[26] J.T. Buck and E.A. Lee, “Scheduling dynamic dataflow graphs with bounded memory using the token flow model,” in *Proc. Int. Conf. Acoustics Speech & Signal Processing (ICASSP)*, pp. 429-432, Apr 1993.

[27] J.T. Buck, S. Ha, E.A. Lee and D.G. Messerschmitt, “Ptolemy: A framework for simulating and prototyping heterogeneous systems,” *Int. Journal of Comp. Simulation*, vol. 4, pp. 155-182, Apr. 1994.

[28] L. Cai and D. Gajski, “Transaction level modeling: An overview”, in *Proc. IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 19-24, Newport Beach, CA, Oct 2003.

[29] C.G. Cassaandras, *Discrete Event Systems: Modeling and Performance Analysis*, Aksen Associates, Boston, MA, 1993.

[30] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya and M. Diaz-Nava, “Component-Based Design Approach for

Multicore SoCs," in *Proc. Design Automation Conf. (DAC)*, pp. 789-794, Jun. 2002.

[31] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," in *Proc. Int. Workshop on Hardware/Software Codesign (CODES)*, Oct. 1993.

[32] J.M. Chang and M. Pedram, "Codex-DP: Codesign of communicating systems with dynamic programming," *IEEE Trans. Comp. Aided Design of Integrated Ckt. and Systems*, vol. 19, no. 7, pp. 732-744, Jul. 2000.

[33] R.E. Crochiere and A.V. Oppenheim, "Analysis of linear digital networks," *Proc. IEEE*, vol. 63, no. 4, pp. 581-595, Apr. 1975.

[34] W.J. Dally and J.W. Poulton, *Digital Systems Engineering*, Cambridge University Press, Cambridge, UK, 1998.

[35] Abhijit K. Deb, A. Hemani, A. Postula and L.S. Nagurney, "The virtual prototyping of an AM chip using grammar based approach," in *Proc. 17th IEEE NORCHIP Conf.*, pp. 283-290, Oslo, Norway, Nov. 1999.

[36] Abhijit K. Deb, A. Hemani, and J. Öberg, "A heterogeneous modeling environment of DSP systems using grammar based approach," in *Proc. 3rd Int. Forum on Design Languages (FDL)*, pp. 365-370, Tubingen, Germany, Sep. 2000.

[37] Abhijit K. Deb, A. Hemani, and J. Öberg, "Grammar based design and verification of an LPC speech codec: A case study," in *Proc. 11th Int. Conf. on Signal Processing Applications and Technology (ICSPAT'2000)*, Dallas, USA, Oct. 2000.

[38] Abhijit K. Deb, A. Hemani, J. Öberg, A. Postula and D. Lindqvist, "Hardware software codesign of DSP systems using grammar based approach," in *Proc. 14th IEEE Int. Conf. on VLSI Design*, pp. 42-47, Bangalore, India, Jan. 2001.

[39] Abhijit K. Deb, J. Öberg, and A. Jantsch, "Simulation and Analysis of Embedded DSP Systems Using MASIC Methodology," in *Proc. IEEE Design, Automation and Test in Europe (DATE) Conf.*, pp. 1100-1101, Munich, Germany, Mar. 2003.

[40] Abhijit K. Deb, J. Öberg, and A. Jantsch, "Simulation and Analysis of Embedded DSP Systems using Petri Nets," in *Proc. 14th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pp. 64-70, San Diego, California, Jun. 2003.

[41] Abhijit K. Deb, A. Jantsch and J. Öberg, "System design for DSP applications using the MASIC methodology," in *Proc. Design, Automation and Test in Europe (DATE) Conf.*, vol. 1, pp. 630-635, Feb. 2004.

[42] Abhijit K. Deb, A. Jantsch and J. Öberg, "System design for DSP applications in transaction level modeling paradigm," in *Proc. Design Automation Conf. (DAC)*, pp. 466-471, Jun. 2004.

[43] J.A. Debardelaben and V.K. Madisetti, "Hardware/software codesign for signal processing system–A survey and new results," in *Proc. 29th IEEE ASILOMAR Conf.*, vol. 2, pp. 1316-1320, Nov. 1995.

[44] E.A. De Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers, "YAPI: Application modeling for signal processing systems," in *Proc. Design Automation Conf. (DAC)*, pp. 402-405, Jun. 2000.

[45] G. De Micheli and R.K. Gupta, "Hardware/Software Co-Design," *Proc. IEEE*, vol. 85, no. 3, Mar. 1997.

[46] S. Dutta, R. Jensen and A. Rieckmann, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," *IEEE Design and Test of Comp.*, vol. 18, no. 5, pp. 21-31, Sep.-Oct. 2001.

[47] B.K. Dwivedi, A. Kumar and M. Balakrishnan, "Synthesis of application specific multiprocessor architectures for process networks," in *Proc. 17th Int. Conf. VLSI Design*, pp. 780-783, Jan. 2004.

[48] R. Dömer, D.D. Gajski and A. Gerstlauer, "SpecC methodology for high-level modeling," in *Proc. 9th IEEE/DATC Electronic Design Processes Workshop*, Monterey, CA, Apr. 2002.

[49] D. Edenfeld, A.B. Kahng, M. Rodgers and Y. Zorian, "2003 Technology roadmap for semiconductors," *IEEE Computer*, vol. 37, no. 1, pp. 47-56, Jan. 2004.

[50] S. Edwards, L. Lavagno, E.A. Lee and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and Synthesis," *Proc. IEEE*, vol. 85, no. 3, pp. 366-390, Mar. 1997.

[51] P. Eles, Z. Peng, K. Kuchcinski and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *Kluwer Journal on Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 5-32, Jan. 1997.

[52] R. Ernst, J. Henkel and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design and Test of Comp.*, vol. 15, no. 4, pp. 65-75, Dec. 1993.

[53] R. Ernst, "Codesign of embedded systems: Status and trends," *IEEE Design and Test of Comp.*, vol. 15, no. 2, pp. 45-54, Apr.-Jun. 1998.

[54] D.D. Gajski, F. Vahid, S. Narayan and J. Gong, "SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 1, pp. 84-100, Mar. 1998.

[55] D. Genin, P. Hilfinger and J. Rabaey, "DSP specification using the Silage language," in *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, vol. 2, pp. 1057-1060, Apr. 1990.

[56] A. Gerstlauer and D.D. Gajski, "System-level abstraction semantics," in *Proc. IEEE Int. Symp. System Synthesis (ISSS)*, pp. 231-236, Oct. 2002.

[57] A. Girault, B. Lee and E.A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Trans. Comp. Aided Design of Integrated Ckt. and Systems*, vol. 18, no. 6, pp. 742-760, Jun. 1999.

[58] R. Grötker, R Schoenen and H. Meyr, "PCC: A modeling technique for mixed control/data flow systems," in *Proc. European Design Automation Conf. (Euro-DAC)*, pp. 482-486, Mar. 1997.

[59] T. Grötker, S. Liao, G. Martin and S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, Norwell, MA, 2002.

[60] P. Le Guernic, M. Le Borgne, T. Gauthier and C. Le Marie, "Programming real time applications with Signal," *Proc. IEEE, Another Look at Real Time Programming, Special Issue*, Sep. 1991.

[61] R.K. Gupta and G. De Micheli, "System level synthesis using reprogrammable components," in *Proc. European Design Automation Conf. (Euro-DAC)*, pp.2-7, Mar. 1992.

[62] R.K. Gupta, C.N. Coelho Jr. and G. De Micheli, "Synthesis and simulation of digital systems containing interacting hardware and software component," in *Proc. IEEE/ACM Design Automation Conf. (DAC)*, pp. 225-230, 1992.

[63] R.K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design and Test of Comp*., vol. 10, no. 3, pp. 29-41, Sep. 1993.

[64] R.K. Gupta and Y. Zorian, "Introducing core-based system design," *IEEE Design and Test of Comp*, vol. 14, no. 4, pp. 15-25, Oct.-Dec. 1997.

[65] S. Ha and E.A. Lee, "Compile-time scheduling of dynamic constructs in dataflow program graphs," *IEEE Trans. Comp.*, vol. 46, no. 7, pp. 768-778, Jul. 1997.

[66] N. Halbwachs, P. Caspi and D. Pilaud, "The synchronous dataflow programming language Lustre," *Proc. IEEE, Another Look at Real Time Programming, Special Issue*, Sep. 1991.

[67] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. of Comp. Programming*, vol. 8, no. 3, pp. 231-274, Jun. 1987.

[68] J. Henkel, "A low power hardware/software partitioning approach for core based embedded systems," in *Proc. Design Automation Conf. (DAC)*, pp. 122-127, Jun. 1999.

[69] J. Henkel and R. Ernst, "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation technique," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 2, pp. 273-289, Apr. 2001.

[70] A. Hemani, J. Öberg, Abhijit K. Deb, D. Lindqvist and B. Fjellborg, "System level virtual prototyping of DSP ASICs using grammar based approach," in *Proc. 10th IEEE Int. Workshop on Rapid System Prototyping (RSP)*, pp. 166-171, Clearwater, Florida, Jun. 1999.

[71] A. Hemani, A. Postula, Abhijit K. Deb, D. Lindqvist and B. Fjellborg, "A divide and conquer approach to system level verification of DSP ASICs," in *Proc. IEEE Int. High Level Design Validation and Test Workshop (HLDVT)*, pp. 87-92, San Diego, California, Nov. 1999.

[72] A. Hemani, Abhijit K. Deb, J. Öberg, A. Postula, D. Lindqvist and B. Fjellborg, "System level virtual prototyping of DSP SOCs using grammar based approach," *Kluwer Journal on Design Automation for Embedded Systems*, vol. 5, no. 3, pp. 295-311, Aug. 2000.

[73] C.A.R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, 1978.

[74] T.B. Ismail, M. Abid and A. Jerraya, "COSMOS: A codesign approach for communicating systems," in *Proc. 3rd Int. Workshop on Hardware/Software Codesign (CODES)*, pp. 17-24, Sep. 1994.

[75] A. Jantsch, P. Ellervee, J. Öberg, A Hemani and H. Tenhunen, "Hardware/software partitioning and minimizing memory interface traffic," in *Proc. European Design Automation Conf. (Euro-DAC)*, pp. 226-231, Sep. 1994.

[76] A. Jantsch, *Modeling embedded systems and SOC's: Concurrency and time in models of computation*, Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[77] M. Jersak, D. Ziegenbein, F. Wolf, K. Richter and R. Ernst, "Embedded system design using the SPI workbench," in *Proc. 3rd Int. Forum on Design Languages (FDL)*, pp. 355-364, Sep. 2000.

[78] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congress'74*, pp. 471-474, Aug. 1974.

[79] G. Kahn and D.B. MacQueen, "Coroutines and networks of parallel processes," *Information Processing '77*, B. Gilchrist, Ed. pp. 993-998, Aug. 1974.

[80] A. Kalavade and E.A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design and Test of Comp.*, vol. 10, no. 3, pp. 16-28, Sep. 1993.

[81] A. Kalavade and E.A. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," in *Proc. 3rd Int. Workshop on Hardware/Software Codesign.*, pp. 42-48, Sep. 1994.

[82] M. Keating and P. Bricaud, *Reuse Methodology Manual*, Kluwer Academic Publishers, Norwell, MA, 1999.

[83] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform based design," *IEEE Trans. Comp. Aided Design of Integrated Ckt. and Systems*, vol. 19, no. 12, pp. 1523-1543, Dec. 2000.

[84] B. Kienhuis, Ed Deprettere, K. Vissers and P. van der Wolf, "An approach for quantitative analysis of application specific dataflow architectures," in *Proc. IEEE*

Conf. Application Specific Systems, Architectures and Processors, pp. 338-349, Jul. 1997.

[85] P.V. Knudsen and J. Madsen, "Integrating communication protocol selection with hardware/software codesign," *IEEE Trans. Comp. Aided Design of Integrated Ckt. and Systems*, vol. 18, no. 8, pp. 1077-1095, Aug. 1999.

[86] T. Kuhn, W. Rosenstiel and U. Kebschull, "Description and simulation of hardware/software systems with Java," in *Proc. Design Automation Conf (DAC)*, pp. 790-793, Jun. 1999.

[87] S. Kumar, A. Jantsch, J.P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä and A. Hemani, "A network on chip architecture and design methodology," in *Proc. IEEE Comp. Society Annual Symposium on VLSI*, pp. 105-112, Apr. 2002.

[88] K. Lahiri, A. Raghunathan and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *IEEE Trans. Comp. Aided Design of Integrated Ckt. and Systems*, vol. 20, no. 6, pp. 768-783, Jun. 2001.

[89] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, A. Sangiovanni-Vincentelli, "A Case Study on Modeling Shared Memory Access Effects during Performance Analysis of HW/SW Systems", in *Proc. Int. Workshop on Hardware-Software Codesign (CODES)*, pp. 117-121, Seattle, USA, March 1998.

[90] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, "Cosimulation-based power estimation for system-on-chip design," *IEEE Trans. Very Lange Scale Integration (VLSI) Systems*, vol. 10, no. 3, pp. 253-266, Jun. 2002.

[91] E.A. Lee and D.G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comp.*, vol. C-36, no. 1, pp. 24-35, Jan. 1987.

[92] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235-1245. Sep. 1987.

[93] E.A. Lee and T.M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773-799. May 1995.

[94] E.A. Lee and A. Sangiovanni-Vincentelli, "A Framework for comparing models of computations," *IEEE Trans Comp. Aided Design of Integrated Ckt. and Systems*, vol. 17, no. 12, Dec. 1998.

[95] M.T.C. Lee, V. Tiwari, S. Malik and M. Fujita, "Power analysis and minimization techniques for embedded DSP software," *IEEE Trans. Very Lange Scale Integration (VLSI) Systems*, vol. 5, no. 1, pp. 123-135, Mar. 1997.

[96] C. K. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee, "Standards for system-level design: Practical reality or solution in search of a question," in *Proc. Design, Automation and Test in Europe (DATE) Conf.*, pp.576-583, Mar. 2000.

[97] S. Liao, S. Tjiang and R. Gupta, "An efficient implementation of reactivity for modeling hardware in the Scenic design environment," in *Proc. Design Automation Conf. (DAC)*, pp. 70-75, Jun. 1997.

[98] P. Lieverse, P. van der Wolf and E. Deprettere, "A trace based transformation technique for communication refinement," in *Proc. 9th Int. Symp. Hardware/Software Codesign (CODES)*, pp. 134-139, Apr. 2001.

[99] B. Lin and S. Vercauteren, "Synthesis of concurrent system interface modules with automatic protocol conversion generation," in *Proc. Int. Conf. Comp. Aided Design (ICCAD)*, pp. 101-108, Nov. 1994.

[100] C. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment" *J. ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[101] H. Liu and D.F. Wong, "Integrated partitioning and scheduling for hardware/software co-design," in *Proc. IEEE Int. Conf. on Comp. Design (ICCD)*, pp. 609-614, Oct. 1999.

[102] A. Mihal, C. Kulkarni, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, C. Sauer, K. Viseers and S. Malik, "Developing architectural platforms: A disciplined approach," *IEEE Design and Test of Comp.*, vol. 19, no. 6, pp. 6-16, Nov.-Dec. 2002.

[103] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and Systems Science*, pp. 375-384, 1978.

[104] G.E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114-117, Apr. 1965.

[105] D. Nadamuni and G. Smith, "Electronic system level design: Challenges on the road ahead," *IEEE Computer*, vol. 36, no. 4, pp. 36-37, Apr. 2003.

[106] M. O'Nils, J. Öberg and A. Jantsch, "Grammar based modeling and synthesis of device drivers and bus interfaces," in *Proc. EuroMicro Conf.* vol. 1, pp. 55-58, Aug. 1998.

[107] M. O'Nils and A. Jantsch, "Operating system sensitive device driver synthesis from implementation independent protocol specification," in *Proc. Design, Automation and Test in Europe (DATE) Conf.*, pp.562-567, Mar. 1999.

[108] K.K. Parhi, *VLSI Digital Signal Processing Systems*, John Wiley & Sons Inc. New York, NY, 1999.

[109] R. Passerone, J.A. Rowson and A. Sangiovanni-Vincentelli, "Automatic synthesis of interfaces between incompatible protocols," in *Proc. Design Automation Conf. (DAC)*, pp. 8-13, Jun. 1998.

[110] J. Peng, S. Abdi and D.D. Gajski, "Automatic model refinement for fast architecture exploration," in *Proc. 15th IEEE Int. Conf. VLSI Design*, pp. 332-337, Jan. 2002.

[111] J. Peterson, *Petri Net Theory and The Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[112] J.M. Rabaey, A. Chandrakasan and B. Nikolic, *Digital integrated circuits: A design perspective*, Prentice Hall, Upper Saddle River, NJ, 2003.

[113] F. Remond and P. Bricaud, "Set top box SOC design methodology at STMicroelectronics," in *Proc. Design, Automation and Test in Europe (DATE) Conf.*, pp. 220-223, Mar. 2003.

[114] K.V Rompaey, D. Verkest, I. Bolsen and H. De Man, "CoWare – A design environment for heterogeneous hardware/software systems," in *Proc. European Design Automation Conf. (Euro-DAC)*, pp. 252-257, Sep. 1996.

[115] J.A. Rowson, "Hardware/software co-simulation," in *Proc. Design Automation Conf. (DAC)*, pp. 439-440, Jun. 1994.

[116] J.A. Rowson and A. Sangiovanni-Vincentelli, "Interface based design," in *Proc. Design Automation Conf. (DAC)*, pp. 178-183, Jun. 1997.

[117] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Trans. Comp. Aided Design of Integrated Ckt. and Systems*, vol. 23, no. 1, pp. 17-32, Jan. 2004.

[118] A. Sangiovanni-Vincentelli, "Electronic-system design in the automobile industry," *IEEE Micro*, vol. 23, no. 3, pp. 8-18, May-Jun. 2003.

[119] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis and M. Sgroi, "Benefits and challenges for platform-based design," in *Proc. Design Automation Conf. (DAC)*, pp. 409-414, Jun. 2004.

[120] A. Seawright, F. Brewer, "Clairvoyant: A synthesis system for production-based specification," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 2 no. 2, pp. 172-185, Jun. 1994.

[121] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, J. Buck, "A system for compiling and debugging structured data processing controllers," in *Proc. Design Automation Conf. (Euro DAC).*, pp. 86-91, Sep. 1996.

[122] M. Sgroi, L. Lavagno and A. Sangiovanni-Vincentelli, "Formal models for embedded systems design," *IEEE Design & Test of Comp.*, vol. 17, no. 2, pp. 14-27, Jun. 2000.

[123] R. Siegmund and D. Müller, "Automatic synthesis of communication controller hardware from protocol specifications," *IEEE Design & Test of Comp.*, vol. 19 no. 4, pp. 84-95, Jul-Aug. 2002.

[124] A. Silberschatz and P.B. Galvin, *Operating System Concepts*, Addison-Wesley Longman Inc., Reading, MA, 1999.

[125] D. Singh, J.M. Rabaey, M. Pedram, F. Catthoor, S. Rajgopal, N. Sehgal and T.J. Mozdzen, "Power conscious CAD tools and methodologies: A Perspective," *Proc. IEEE*, vol. 83, no. 4, pp. 570-594, Apr. 1995.

[126] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst and J. Teich, "FunState—An internal design representation for codesign," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 4, pp. 524-544, Aug. 2001.

[127] P. Thoma, "Automotive electronics–A challenge for systems engineering," in *Proc. Design, Automation and Test in Europe (DATE) Conf.*, pp. 4, Mar. 1999.

[128] F. Vahid and D.D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 3, pp. 459-464, Sep. 1995.

[129] S. Vernalde, P. Schaumont and I. Bolsens, "An object oriented programming approach for hardware design," in *Proc. IEEE Comp. Soc. Workshop*, pp. 68-73, Apr. 1999.

[130] J. Voelcker, "Top 10 Tech car," *IEEE Spectrum*, vol. 41, no. 3, pp. , Mar. 2004.

[131] P. van der Wolf, P. Lieverse, M. Goel, D.L. Hei and K. Vissers, "An MPEG-2 decoder case study as a driver for a system level design methodology," in *Proc. 7th Int. Workshop on Hardware/Software Codesign (CODES)*, pp. 33-37, May 1999.

[132] W. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE*, vol. 82, no. 7, pp. 967-989, Jul. 1994.

[133] W. Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 2, pp. 218-229, Jun. 1997.

[134] W. Wolf, "A decade of Hardware/Software Codesign," *IEEE Computer*, vol. 36, no. 4, pp. 38-43, Apr. 2003.

[135] W. Wolf, "The future of multiprocessor Systems-on-Chips," in *Proc. Design Automation Conf. (DAC)*, pp. 681-685, Jun. 2004.

[136] L.R. Zheng, *Design, analysis and integration of mixed-signal systems for signal and power integrity*, Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, May 2001.

[137] D. Ziegenbein, R. Ernst, K. Richter, J. Teich and L. Thiele, "Combining multiple models of computation for Scheduling and allocation," in *Proc. 6th Int. Workshop on Hardware/Software Codesign (CODES)*, pp. 15-18, Mar. 1998.

[138] D. Ziegenbein, K. Richter, R. Ernst, J. Teich and L. Thiele, "Representation of process mode correlation for scheduling," in *Proc. Int. Conf. Comp. Aided Design (ICCAD)*, pp. 54-61, Nov. 1998.

[139] D. Ziegenbein, K. Richter, R. Ernst, J. Teich and L. Thiele, "SPI–A system model for heterogeneously specified embedded systems," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 379-389, Aug. 2002.

[140] J. Öberg, A. Kumar, and A. Hemani, "Grammar-based hardware synthesis from port size independent specifications," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 2, pp. 184-194, Apr. 2000.