Formal Methods for Intellectual Property Composition Across Synchronization Domains

Syed Mohammed Suhaib

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Dr. Sandeep K. Shukla, Chair
Dr. Dong S. Ha, Committee Member
Dr. Michael S. Hsiao, Committee Member
Dr. Pushkin Kachroo, Committee Member
Dr. James E. Shockley, Committee Member

August 29, 2007
Blacksburg, Virginia

Keywords: Intellectual Property Composition, Latency Insensitive Designs, Globally Asynchronous Locally Synchronous Designs, Correct-by-Construction Protocols

Formal Methods for Intellectual Property Composition Across Synchronization Domains

Syed Suhaib

(ABSTRACT)

A significant part of the System-on-a-Chip (SoC) design problem is in the correct composition of intellectual property (IP) blocks. Ever increasing clock frequencies make it impossible for signals to reach from one end of the chip to the other end within a clock cycle; this invalidates the so-called synchrony assumption, where the timing of computation and communication are assumed to be negligible, and happen within a clock cycle. Missing the timing deadline causes this violation, and may have ramifications on the overall system reliability. Although latency insensitive protocols (LIPs) have been proposed as a solution to the problem of signal propagation over long interconnects, they have their own limitations. A more generic solution comes in the form of globally asynchronous locally synchronous (GALS) designs. However, composing synchronous IP blocks either over long multicycle delay interconnects or over asynchronous communication links for a GALS design is a challenging task, especially for ensuring the functional correctness of the overall design. In this thesis, we analyze various solutions for solving the synchronization problems related with IP composition. We present alternative LIPs, and provide a validation framework for ensuring their correctness. Our notion of correctness is that of *latency equivalence* between a latency insensitive design and its synchronous counterpart. We propose a trace-based framework for analyzing synchronous behaviors of different IPs, and provide a *correct-by-construction* protocol for their transformation to a GALS design. We also present a design framework for facilitating GALS designs. In the framework, Kahn process network specifications are refined into *correct-by-construction* GALS designs. We present formal definitions for the refinements towards different GALS architectures. For facilitating GALS in distributed embedded software, we analyze certain subclasses of synchronous designs using a Pomset-based semantic model that allows for desynchronization toward GALS.

This project received support from CCF-0417340 (CRCD), CCF-0237947 (CAREER), and CCF-0702316 (NSF).

To Amma, Abba, Ayya, Ahad Bhai, and Alina

To my colleagues at FERMAT Lab.

To those affected by 04/16/2007.

Acknowledgments

I would like to acknowledge various people with whom I was directly/indirectly associated with during the course of my PhD.

First and foremost, I would like to acknowledge my mentor, Dr. Sandeep Shukla for his guidance, motivation, encouragement, support, and friendship throughout my stay at the FERMAT Lab. He has been a great inspiration and an educator throughout my PhD. I would like to thank Dr. Dong Ha, Dr. Michael Hsiao, Dr. Pushkin Kachroo and Dr. James Shockley for serving on my PhD committee. I would also like to thank Michael Kishinevsky, Sava Krstic, John O' Leary, Connie Heitmeyer, Elizabeth Leonard, and Myla Archer for educating me at my internships at Intel and Naval Research Lab. I would like to acknowledge the support received from CRCD grant CCF-0417340, CAREER grant CCF-0237947, and NSF grant CCF-0702316, which provided funding for the work reported in this dissertation.

I would like to thank my friend, roommate and colleague Deepak Mathaikutty for numerous hours of discussions on/off my PhD topic, and for being patient and supportive through these discussions. I would also like to thank David Berner for discussions on latency insensitive protocols. Special thanks to all my roommates, friends and colleagues from Virginia Tech: Debayan Bhaduri, Hiren Patel, Gaurav Singh, Sumit Ahuja, Bijoy Jose, Animesh Patcha, Habeeb Abdullah, Madhup Chandra, Nimal Lobo, and Karel Oudheusden.

I would also like to acknowledge Ambareen and family, Ahmad uncle and family, Farhan bhai and family, and Laurita and family for their encouragement and support.

In addition to the above, I would like to thank my family for their unconditional love, support and stimulating me in pursuing my education and research activities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The deep-submicron (DSM) process technology has enabled higher transistor densities on a chip providing higher integration, more gates, more tasks, and more parallelism; however, the design engineers are struggling with the tools and design methodologies to create complex designs that can take advantage of this increasing trend. As a result, the gap between what is possible to implement in silicon, and what is possible to design within the time-to-market time limit is increasing; hence, creating a productivity crisis. Reusing pre-existing blocks such as memories, processor cores, and dedicated hardware blocks chosen from an intellectual property (IP) library seems to be one of the ways to mitigate the problem of productivity crisis and shortening time-to-market cycles. The main goal of design reuse is to accelerate the product development time by rapidly integrating pre-verified blocks together, instead of designing the blocks from scratch. Designing IPs for reuse has its own cost, typically two to three times the cost of designing an IP for a single use [196]. However, when the IP is reused, it takes much less design effort than restarting from scratch. As a result, IP reuse in the design process has grown from 10% to 90% from 1991 to 2001 [196].

Plethora of products such as cell phones, wireless computers, and other multifunction products such as PDAs, MP3 players, and high resolution digital cameras that process multi-million pixels per second are seen in the market today, where each of these are driven by an embedded system-on-a-chip (SoC) that integrates multiple IP blocks. A typical SoC consists of microcontrollers, microprocessors or digital signal processors (DSPs), a wide range of memory blocks including ROM, RAM, with clock generators including oscillators and phase-locked loops (PLLs). Furthermore, it may also include peripherals such as counters, timers, reset generators, analog-to-digital converters (ADCs), digital-to-analog converters (DACs), and other computational blocks with external interfaces based on industry standards such as USB, ethernet, firewire, etc. Figure 1.1 shows a typical example of an SoC.

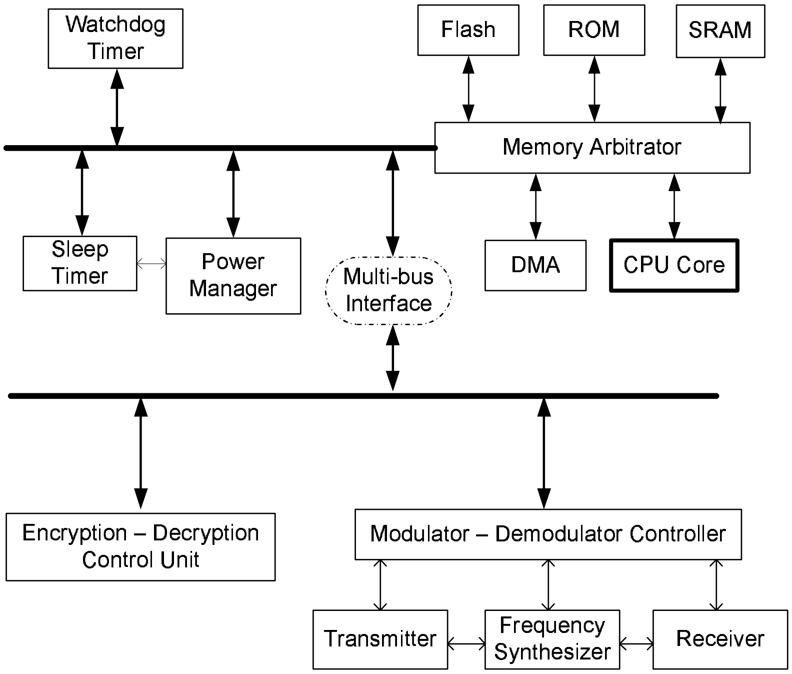


Figure 1.1: A Typical SoC

1.1 IP Classification

In electronic design, IP core stands for a reusable unit of logic, cell or chip layout design which belongs to one party. Earlier, IP cores were mostly functional computation blocks built within the same company for design generations. In today's market [2, 8, 11, 14, 25], IP cores are marketed as off-the-shelf optimized blocks that can be integrated with other blocks. One main reason behind this easy integration is that the IPs are made flexible, customizable and reconfigurable.

IPs can be classified in various forms: soft, hard, and firm. Soft IPs can be modified at the functional-level based on customer requirements. Such IPs are delivered as either netlists or synthesizable cores in the form of a hardware description language such as Verilog or VHDL [2, 11]. On the other hand, in hard IPs, the functionality cannot be modified by the customers. These IPs are best for play-and-plug applications, and are less portable and flexible than the soft IP cores. Such IPs mainly involve analog and mixed-signal logic physical descriptions such as signal processors, etc. Some examples of hard IP cores include ADC, DAC, and PLLs which are offered by vendors like IBM, and TI [8, 22]. Firm IPs fall in the middle of soft and hard IPs. They offer little risk in terms of timing closure in place and route domain [193]. Depending of the IP vendor, parts of a firm IP are delivered as fixed layouts and the remaining parts are configurable. Table 1.1 shows further comparisons between these IP types.

These IPs are optimized for timing, and are provided with information for optimal clock rates for their execution. This dissertation primarily focuses on specific timing related issues with IP

IP Classes	Flexibility	Protection	Predictable	Portability	Technology
Soft IP	Very flexible	Weak	Unpredictable	Unlimited	Independent
Firm IP	Flexible	Medium	Predictable	Library Mapping	Generic
Hard IP	Inflexible	Good	Very predictable	Process Mapping	Fixed

Table 1.1: IP Classifications

composition which we discuss in the following sections.

1.2 IP Composition

With the increasing trend of IP reuse, IP composition has gained significant importance in the design of SoCs. *IP composition signifies the task of bringing together reusable IPs that perform different functionalities, and make them complement each other to achieve the desired behavior.* The task of IP composition for an SoC requires a lot of effort in terms of selection of appropriate IPs from various IP libraries, and making design choices with respect to design languages, agreeable behaviors, interfacing between clock domains, model of computation, bus interface protocols, etc [135, 152, 124, 125, 49, 136, 40].

1.2.1 Trends in IP Composition

IP Standardization

To increase flexibility and reusability of IPs between different tools, many companies such as Mentor Graphics [11], ARM [2], Cadence [4], Synopsys [21], Philips [14], etc, have formed an alliance called the SPIRIT consortium [19]. The main goal of this consortium is to create a standard for IP representation such that IPs from different sources can be used in a unified framework. Hence, an XML-based IP representation is invented with the aim of sharing data in an unambiguous and concise way. It also includes information that allows capturing and reusing all the cross domain information required for IP reuse. The documentation of the IPs is recorded in an XML data book that contains component information on hardware descriptive languages (HDLs), interfaces, software, etc. Moreover, generators are provided with the XML formats that generate design output files such as HDLs, software, design documents, simulator, etc as desired by the designer.

IP Interfacing

Virtual Socket Interface Alliance (VSIA) [25] and Open Core Protocol International Partnership (OCP-IP) [12] have proposed interface standards (VCI and OCP, respectively) to ease integration of blocks. In these standards, the compliant blocks are designed based on a bus-independent standardized interface. A generic interconnection protocol can then be used for communication between these different IPs. One such example is of Sonics Inc. [17] that provides a family of interconnect products to facilitate composition of IPs for an SoC design. They provide wrappers that adapt to bus-independent OCP sockets for its network bus. CoreConnect^{TM} [5] from IBM and AMBA [1] from ARM are some of the publicly available bus architectures that are used for bus-based composition techniques.

The communication between IP blocks can also be in a point-to-point manner. Open Core Protocol (OCP) from Sonics Inc. [17] is one such point-to-point protocol that can be used for connecting two IPs together in a master-slave manner. The master sends command requests to slave, and the slave responds to them. This communication occurs on unidirectional signals.

There are also Network-on-chip (NoC)-based composition techniques where the communication occurs based on packet-switching [40]. The ideas are borrowed from the traditional networking paradigm and applied to an SoC. The communication among IPs is done using switches, network interfaces and channels. The switching in NoC provides a natural pipelining for communication. Various network topologies such as honeycomb or mesh can be implemented [134, 126, 137]. The rules for communication are defined by the transport layer and the packets are accordingly routed through the switches. NoCs have various advantages with respect to communication, power, design productivity and heterogeneity of functions over traditional SoCs [136].

Composition Frameworks

Various frameworks for composition of IPs for creating SoC designs have also been proposed [5, 163, 225, 199, 100, 9].

IBM's SoC framework called IBM Blue Logic^{TM} consists of a pre-verified IP core library that is designed to work with its fixed bus architecture called CoreConnect [5]. Although the IP cores of the library are designed interface-ready with the bus architecture, there are still several issues that the designer has to consider. From assigning priorities for bus accesses to defining correct address mappings for memory accesses are some of the issues highlighted in [49].

In [163], a metamodeling-driven component composition (MCF) framework is provided for automated IP selection from a library of SystemC IPs. The framework enables design space exploration using concepts of metamodeling, type definitions, reflection & introspection for IP selection [163].

The Liberty Simulation Environment (LSE) [9, 225] is a modeling environment that allows rapid construction of high-level simulation models. Based on a concurrent-structural model specification language, LSE supports low-overhead use and construction of reusable blocks [225]. It also provides an abstraction that simplifies the specification of timing control.

For software IPs, a component compositional framework called "*XCompose*", based on XML, has been proposed for composing software blocks. The framework provides flexibility, extensibility, reusability and correctness of composition. It is based on the hypothesis that a set of primitive composition operators can be combined to produce arbitrary complex compositions and composition patterns, thereby providing flexibility and extensibility. Composition pattern templates are provided to save the composition patterns for later reuse. The composition is done using *compositional operators* and *glue logic* [221].

For embedded system design, a framework called “BALBOA” has been proposed that implements a compositional approach to high-level modeling [199, 100]. The BALBOA component composition environment is a layered environment that provides a component model with *introspection* and partial typing capabilities [199]. Components are composed dynamically by encapsulating them with automatically generated wrappers that implement the compositional rules, dynamic type determination and type inference algorithms. This framework is used to build system models with an architectural perspective.

1.2.2 Issues with IP Composition

A significant part of the SoC design problem is the correct composition of these existing IP blocks [58]. IP blocks obtained from different vendors, or even different design groups from the same company, are heterogeneous in several aspects. They can differ in design domains, programming languages, models of computations, abstraction levels, bus communication interfaces, and various other features. We discuss some of these issues below:

Interface Compatibility Issues

As discussed earlier, one can make various interface design choices of how the IPs interact. However, IPs must be appropriately connected to its output. For example, ensuring data and address signals are correctly connected, ensuring bus connections are appropriately made with respect to bus widths (32, 64, or 128-bit buses), ensuring consistency with regards to name matching at interfaces, etc. Furthermore, priorities associated with pin matching, bus access, etc, must be made correctly.

Behavioral Compatibility Issues

It is important to ensure behavioral compatibility of IPs being composed. This can be done by adding appropriate glue logic to ensure that the data exchanged between IPs is in appropriate formats. For example, a producer IP may produce data with its units as seconds, whereas, the IP consuming this data may be expecting data as milliseconds. It is also important to ensure that domains of IPs that are being composed are same. In the case of domain mismatch, it may happen that a data value produced by one IP cannot be consumed by its destination IP. Hence, the protocols of the IPs being composed must match.

Next, we discuss the timing related issues with IP composition.

Timing-Related Issues

In this dissertation, our focus is on the timing related issues:

Long interconnect issues: In the industry, most of the design tools and methodologies advocated are synchronous in nature. By synchronous, we mean that the communication and computation are assumed to be instantaneous[1]. Hence, the use of such tools for creating SoCs by composing different IPs together imposes the rule that the communication between the IPs is instantaneous. However, when such SoCs are implemented on hardware, the ever increasing clock frequencies invalidate the synchrony assumption on the interconnects between IP blocks [58, 70]. The clock period becomes too small for two interfacing IPs to exchange data across interconnects within such small clock period. This is because some interconnects are longer than the distance a signal propagates during a single clock cycle [71]. As a result, the delay on such interconnects renders the output of the design to be incorrect. For example, consider that the SoC given in Figure 1.1 is designed to be synchronous and driven by a single clock. When this design is synthesized onto hardware, it may so happen that the system is unable to process the data within the same clock cycle due to the long communication delays between the blocks. It has been predicted that for DSM designs, a signal will need more than ten clock cycles to traverse the entire chip area [105, 164]. Similar issues have been seen in the design of high-performance microprocessors. One such example is of the hyperpipelined NetBurst microarchitecture of Intel Pentium 4 processor [117, 127], where pipeline stages are dedicated to exclusively handle wire delays. It contains two *drive-stages* that are used exclusively to propagate the signal across the chip.

Clock synchronization issues: Another major problem with composing IPs involves ensuring proper synchronization between them. The delay variations in the clock signals reaching different IPs on a chip causes undesirable behavior. There are various types of clock delay variations: skew, jitter, drift, etc. which are realized only after the implementation on architectures. The *clock skew* is a phenomenon considered when the clock signal arrives at different times for different blocks. This may be caused due to unequal wire length, unequal load, or IR drop. There can be two forms of violations that can be caused by clock skew: *hold violation* and *setup violation*. The *hold violation* occurs when the destination flip-flop receives the clock tick later than the source flip-flop. This happens when the data signal arrives at the destination flip-flop before the clock tick, destroying the previous data that should have been clocked through. The *setup violation* occurs when the new data is not set up and stable before the next clock tick arrives at the destination flip-flop. This happens when the destination flip-flop receives the clock tick earlier than the source flip-flop [106, 157, 220]. Typical high-performance designs such as microprocessors for general purpose computers are driven by a single clock [75]. Given such high clock frequencies, designers traditionally target clock skews of about 10% of the clock period [133, 142, 146, 149]. However, long wire delays and variations in buffer delay make these targets challenging. Another problem

[1] Synchrony assumption.

seen in the synchronous SoCs is *clock jitter* which arises from inaccuracies in the source oscillator, or drifting of the phase lock loop (PLL). *Jitter* mainly applies to a cycle-to-cycle variation from one period to the next period of the clock. This may be caused due to the noise through capacitive and inductive coupling of wires [192]. *Clock drift* is another issue seen in synchronous circuits that refers to slow changes that occurs when two clocks are not driven by exact same speed. These drifts may accumulate to larger values, and are mainly caused due to slow changes in voltage or temperature [106].

In section 1.4, we pose the problems that we are solving in this dissertation, and discuss why the existing solutions for these timing related issues are inadequate. In the following section, we consider timing related issues in multi-clock domains.

1.3 Clock Domains

A *clock domain* refers to a part of the design that is driven by a clock. The clock domains can be classified into *single-clock* or *multi-clock* [114]. The *single-clock* domain has all its blocks driven by a single clock. On the other hand, blocks in the *multi-clock* domain have multiple clocks, and they can be further classified into two categories: (1) the designs can have same frequencies but different phases (also called *multi-synchronous* domain), or (2) the designs can have different frequencies which can either include a *central control* system with dynamic voltage scaling (DVS) or a system with an *autonomous control* with globally asynchronous locally synchronous (GALS) domains or asynchronous domains. This taxonomy [114] is shown in Figure 1.2.

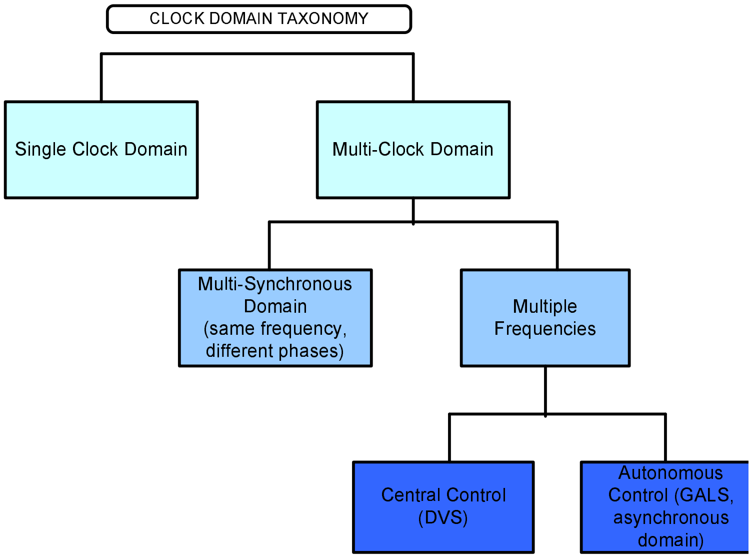

CLOCK DOMAIN TAXONOMY
Single Clock Domain
Multi-Clock Domain
Multi-Synchronous Domain (same frequency, different phases)
Multiple Frequencies
Central Control (DVS)
Autonomous Control (GALS, asynchronous domain)

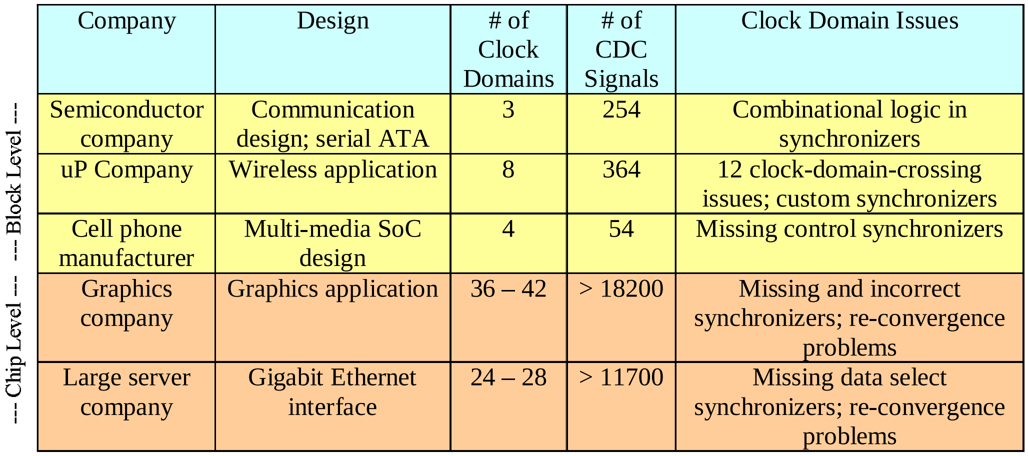
Figure 1.2: Taxonomy of Clock Domains

Most IPs assembled onto an SoC belong to different clock domains. This may be due to interfacing with PCI and USB components, which are external components. Another reason for using multi-clock domains on an SoC may be due to the distribution problem of a single clock over the entire large chip (discussed in Section 1.2.2). Furthermore, multi-clock SoCs have been used for creating efficient designs with respect to power, throughput, etc [231, 133].

1.3.1 Trends in Clock Domains

On-chip clocks in today's SoCs are in the range from 200 MHz - 500 MHz. The numbers of clock domains on a single SoC are in the range of 5 to 15. A signal that originates in one clock domain, and passes to a different clock domain to be sampled by a register is called a *clock-domain crossing* (CDC) signal. It is estimated that in modern SoCs, an order of 10^4 signals cross clock domains on a single chip [125]. Therefore, incorrect handling of CDC signals is one of the major causes of re-spin in multi-clock SoCs [124].

Figure 1.3 shows the clock domain problems faced by various industries [124]. Traditionally, at the block level, not many clock domains are involved. There are about 3 - 10 clock domains with the number of CDC signals in hundreds. However, at the chip level, more complex SoCs are being designed with CDC signals in tens of thousands. Most of the errors are due to missing or incorrect *synchronizers* and protocols (We discuss these in the following sections.).



	Company	Design	# of Clock Domains	# of CDC Signals	Clock Domain Issues
--- Block Level ---	Semiconductor company	Communication design; serial ATA	3	254	Combinational logic in synchronizers
	uP Company	Wireless application	8	364	12 clock-domain-crossing issues; custom synchronizers
	Cell phone manufacturer	Multi-media SoC design	4	54	Missing control synchronizers
--- Chip Level ---	Graphics company	Graphics application	36 – 42	> 18200	Missing and incorrect synchronizers; re-convergence problems
	Large server company	Gigabit Ethernet interface	24 – 28	> 11700	Missing data select synchronizers; re-convergence problems

Figure 1.3: Clock-Domain Problems

1.3.2 Clock Domain Crossing Issues

Ensuring correct communication between different clock domains is one of the major concerns in multi-clock SoC designs. To ensure correct communication between domains running on multiple clocks, we need a robust communication protocol. This protocol must ensure that when IPs running on independent clocks are composed together, then the resultant behavior is the intended behavior. These protocols act as a gluing logic across the domains including long interconnects. Due to lack of tools for multi-clock designs, it is difficult to ensure the correctness of such protocols. Furthermore, artificial deadlocks can be introduced as an artifact of the communication protocols [112]. Artificial deadlocks occur when small size buffers are chosen between synchronous blocks. Consider the example shown in figure 1.4, with synchronous components a, b, c and d. We assume that the communication protocol works as follows: each component can execute only if it has data on its corresponding inputs and its destination components are ready to accept data.

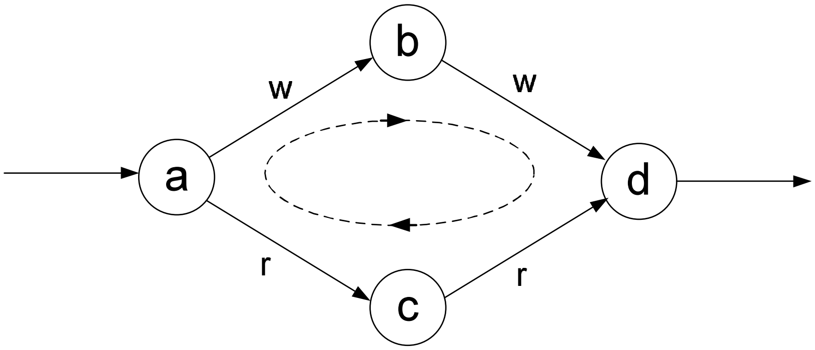


Figure 1.4: An artificial deadlock

The signals from a to b and b to d are marked with w, indicating that components a and b are blocked for writing (output buffers full) , whereas signals from a to c and c to d are marked with r, indicating that the components c and d are blocked on reads (input buffers empty). This means that d has data from b, but is waiting for data from c. Also, c is waiting for data from a, however, a cannot write because is it blocked from b. Therefore, there is an artificial deadlock.

Secondly, every signal that crosses the clock domain require some from of synchronization. Synchronizers are used to ensure that *metastability* is avoided. All storage elements are susceptible to *metastability*, and mainly those at the boundary of the clock domains. Traditional verification practices (mainly simulation techniques) are not sufficient to detect such errors.

Metastability: The correct operation of a flip-flop depends on a stable input for a certain period of time before and after the clock edge. This window is made up of the *setup time* and the *hold time* (discussed earlier in Section 1.2.2). If the setup and hold requirements are satisfied, then a correct output will be realized at a valid output level. If these requirements are not satisfied, the flip-flop may take much longer to reach a valid output level. This happens mostly during an interaction between two elements that are from different clock domains. To address this issue, circuits must be designed such that the *metastable* signal can settle to a stable value. *Synchronizers* are effective in handling *metastability* if applied correctly. However, one has to be careful in placing and using the synchronizers appropriately. Various types of synchronizers are used in the industry. A commonly used synchronizer is a two D-flip flop synchronization circuit that allows ample time for settling of a complete cycle [116, 99]. Other techniques used in the industry include Data-MUX synchronizers, FIFOs, handshakes, reset, pulse, etc, where each scheme has a specific structure and rules associated with it [125, 106, 107]. To avoid metastability, one can also ensure that the signal is not sampled when it is changing. In this dissertation, we work at the protocol level, and hence are not concerned with metastability, which is mainly a circuit level problem. We assume that at the circuit level, the above discussed solutions can be employed.

Errors also occur when there is a presence of combinational logic in the synchronization path. Since combinational logic is prone to glitches, a glitch can be captured as a valid value by the receiving synchronizer, and propagated to the design [125].

Next, we discuss another multi-clock domain model where synchronous components communication asynchronously.

1.3.3 Globally Asynchronous Locally Synchronous Designs

For composing IP blocks with different clock speeds, designing towards GALS domain is another solution. GALS designs provide the idea of making communication asynchronous between synchronous IP blocks. Each locally synchronous island in a GALS design can be encapsulated in an wrapper, which facilitates inter-island asynchronous communication. In most cases, these wrappers also generate the clocks for their own respective blocks eliminating the clock skews, which are present in large synchronous designs. Various strategies can be employed to implement GALS designs. Pausible clocking [232, 233, 202] is one such solution where clocks are generated locally, and appropriate logic is employed to stretch or pause the blocks due to full or empty signals at their corresponding inputs/outputs. IP blocks can also be connected using asynchronous interfaces [78] such as mixed signal FIFOs [79]. For multi-synchronous designs [114], loosely synchronous interfaces [76, 75, 167] can be used. However, there is a lack of GALS frameworks and design tools which makes it difficult to ensure the correctness of such designs. In this dissertation, we propose solutions to create correct-by-constructions GALS designs.

There is another notion of GALS besides the one in hardware domain. This GALS notion pertains to the distributed real-time embedded software. When one designs a real-time software for a particular embedded platform, timing concerns can be abstracted by synchrony assumption, and the functionality of the software component can be modeled with the so called 'synchronous languages' such as Esterel [56], SIGNAL [32] or Lustre [123], prevalent in French embedded systems community. Correctness proofs of the functional model of such software can be established under synchrony hypothesis, analysis of the schedulability of computation, event acceptance and generation can be done in formal calculus, and then real-time embedded code can generated which is provably *correct-by-construction* with timing guarantees. In the following section, we discuss Polychrony, a synchronous language, that enables such *correct-by-construction* designs.

1.3.4 Polychrony

In real-time embedded software, designs are run on distributed nodes. The synchrony hypothesis, which is assumed for communication between these nodes, breaks down because the communication of events between different blocks is no long fast enough to justify such assumptions. Hence, GALS designs are being implemented to resolve this issue of distributed code generation, where nodes communicate via asynchronous protocols.

Polychronous MoC [122] facilitates the description of systems in which blocks obey to multiple clock rates, ideal for real-time embedded software. The reason for such multiple clock rates is that such systems interact with the environment, and the rate at which environment generates values are not necessarily known unless constraints are imposed on the environment, which often is not possible. Polychrony also provides a mathematical foundation to a notion of refinement, and the ability to model a system from the early stages of its requirement specifications to the late stages of its synthesis and deployment. Polychrony favors the progressive design of correct by construction systems by means of well-defined model transformations that preserve the intended semantics of early requirement specifications and eventually provide a functionally correct deployment on target architectures. Polychrony is a concurrency model where activities happen at their own pace except when explicit synchronizations are required. Polychronous programs have been studied to have various characteristics [43, 122, 41] that help in analyzing designs that can be correctly desynchronized [43, 122] into globally asynchronous locally synchronous designs.

1.4 Problems Addressed by this Dissertation

In this dissertation, we target three main problems that are:

Problem 1

Definition: Lack of validation frameworks for latency insensitive designs.

Motivation: Several approaches have been proposed to deal with the problem of long latencies in global physical chip interconnects in an SoC design. One is to create packet based Networks-on-Chip (NoC), also targeted at interconnect latencies [40]. However, they require sophisticated protocols to enable correct communication. Another approach is the application of latency insensitive protocols (LIP) to the synchronous designs to make them latency insensitive (LI) [71, 68, 69, 70]. The latency insensitive protocols require encapsulation of all modules with wrapper logic and insertion of extra memory elements along the long interconnects based on their cycle delays. These protocols are primarily applicable to hard IPs. For soft IPs, latency insensitive protocol called synchronous elastic flow (SELF) was developed [89, 88, 148]. The correctness of such protocols needs to be established to ensure that their application preserves the desired functionality of the design. There are many ways one can ensure correctness of latency insensitive systems. Dynamic validation is one of the easiest ways to check for correct traces as it helps in flushing out protocol

design errors. However, such validation only covers certain input sequences. Using formal verification techniques such as model checking is a more desirable validation mechanism, where the design can be modeled formally and validated based on its properties. However, model checking may turn out to be extremely resource consuming. Another way to confirm the correctness of such an implementation is to mathematically formalize it, as done in [70]. But mathematically proving the equivalence of two systems is a challenging task and not beyond mistakes. It requires complex mathematical proofs that are not straightforward to follow by others who want to confirm them; hence every new variation of LIPs cannot be validated easily using mathematical proof techniques.
Solution Proposed: We present validation techniques and frameworks for ensuring correctness of latency insensitive designs [206, 207, 214]. In our framework, we model the latency insensitive design along with its synchronous counterpart, and check for latency equivalence [207, 68]. Simulation-based validation as well as model-checking based techniques can be used for checking correctness of the protocols using our framework. This approach is well suited for hard IPs, and assume that these IPs satisfy the *stallable property* [68] which states that the block can be stalled for arbitrary amount of clock cycles without losing its internal state. For soft IPs, we verified the correctness of the SELF protocol in our framework as well as with respect to the definitions of the elastic machines [148]. We also present two LIPs: relay-station based and bridge-based. Our relay station based protocol is a simplification of an existing protocol presented in [68], and requires relay-stations to be placed at strategic points along the long interconnects. This may require multiple place and route iterations. To alleviate these iterations, we present an alternate protocol called bridge-based protocols where the relay stations are merely replaced by wires with placement logic blocks placed at the interface of the transmitter and receiver components.

Problem 2

Definition: Lack of complete frameworks and design tools for creating correct GALS designs.

Motivation: Complete transformation of a synchronous design to a latency insensitive design requires prior knowledge of the cycle delays on the long interconnect. This is because existing protocols require pipelining the long interconnects by adding appropriate number of storage elements on them. Moreover, the information of the long interconnects and the number of cycles required for data to communicate through them cannot be known until after the final layout of the design. Another problem with the latency insensitive designs is that they were mainly proposed for single clock designs. Given these issues, we consider composition towards a GALS design, where the communication on the long interconnects is handled by an asynchronous communication protocol. However, there is a lack of design methodologies and frameworks to facilitate GALS designs. In most cases, GALS designs are constructed using ad hoc methods, where synchronous blocks are encapsulated with some wrapper logic and communication is either handshake-driven or bounded FIFOs are used. It is important to ensure that the communication protocol between these IPs is correct and their composition yields the desired behavior. This becomes difficult due to lack of validation tools. Furthermore, artificial deadlocks can be introduced as an artifact of the communication protocols. Artificial deadlocks occur when small size buffers are chosen between synchronous blocks.
Proposed Solution: We propose two different solutions for creating GALS designs. Firstly, we

propose a formal framework, where an SoC design composed of multi-clock IPs is analyzed for “single-activation” property [211, 208], which means that for all IPs of the system, either all their the inputs and outputs have valid events at every clock tick or no events occur on any of them. In our framework, if the property of “single-activation” holds for a design, then it can be transformed to a behavioral equivalent GALS. Two behavioral-equivalent designs produce the same functional outputs for same inputs. We show behavioral-equivalence, by showing theoretically that systems with single activation are latency equivalent to their GALS design. The single-activation property helps in identifying designs that can be transformed to GALS with barrier synchronizing inputs and avoiding complex handshake protocols. Now, if *single activation* holds for a design, then it can be transformed to a GALS design where its components can execute on independent clocks. This GALS design is correct with respect to its behaviors, and hence does not need to be verified again. Secondly, we propose a design framework for facilitating correct-by-construction of GALS designs. In this design framework, we start from a description as the Kahn Process Network (KPN) [138] model, and perform architectural exploration, and propose appropriate protocols and their refinements for GALS architectures. We theoretically show that our refinements from a KPN model yield a correct-by-construction GALS design. In our correct-by-construction refinements for GALS architectures, we assume that metastability is handled at the circuit level.

Problem 3

Definition: Lack of frameworks for analyzing “true concurrency” semantics for Polychrony.

Motivation: The existing semantic model of Polychrony [122] is based on the tagged-signal model [151, 152]. For analyzing the properties of Polychrony that creating GALS designs, we feel that the tags are unnecessary artifacts, and complicate the semantic theory unduly. A deterministic program may have multiple possible synchronous interpretations. In this context, a synchronous interpretation means that an observer is observing the behavior with respect to successive reactions. Multiple possible synchronous interpretations can occur when the environment generates independent inputs at multiple unrelated rates. As a result, if sequential code is generated from such a polychronous specification, multiple possibilities will exist, and a choice of one over another will not be correct without having extra constraints on the environment. So if one is interested only in the class of programs that has correct and unique sequential implementation, one would be interested in the class of endochronous specifications [43, 122, 41].

Proposed Solution: We propose a true concurrency semantic model of Polychrony using partially ordered multisets (Pomsets) [131, 169, 189]. *True concurrency semantics* are useful to analyze multiple behaviors that occur simultaneously, which in contrast is different from *interleaving semantics*, where such behaviors occur in an interleaved manner (cf. models in [130, 166, 86]). This formulation of Polychrony using Pomsets closes the gap between synchrony and asynchrony by giving a uniform characterization of both synchronous and asynchronous observations by considering the causality and functional dependency structure of a Pomset [210, 215].

1.5 Author's Publications

1. S. Suhaib, D. Mathaikutty, and S. Shukla. Data flow architectures for GALS*, in Proceedings of Formal Methods on Globally Asynchronous Locally Synchronous Designs*, 2007 [213].

2. S. Shukla, S. Suhaib, D. Mathaikutty, and J.-P. Talpin. On the Polychronous Approach to Embedded Software Design, *Next Generation Design and Verification Methodologies for Distributed Control Systems*, pp. 261-274, 2007 [198].

3. S. Suhaib, D. Mathaikutty, and S. Shukla. Polychronous methodology for system design: A true concurrency approach, *in Proceedings of High Level Design Validation and Test Workshop (HLDVT)*, 2006 [210].

4. S. Suhaib, D. Mathaikutty, S. Shukla, D. Berner, and J.-P. Talpin, "A Functional Programming Framework for Latency Insensitive Protocol Validation," Electr. Notes Theor. Comput. Sci., vol. 146, no. 2, pp. 169188, 2006. (Presented at Formal Methods on Globally Asynchronous Locally Synchronous Systems, 2005) [214].

5. S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Validating families of latency insensitive protocols, *in Proceedings of High Level Design Validation and Test Workshop (HLDVT)*, 2005 [206].

6. S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Validating families of latency insensitive protocols, *IEEE Transactions on Computers*, vol. 55, no. 11, pp. 13911401, 2006 [207].

7. S. Suhaib, D. Mathaikutty, and S. Shukla. A trace based framework for validation of SoC designs with GALS systems, *in Proceedings of IEEE International SOC Conference*, 2006 [211].

8. S. Suhaib, D. Mathaikutty, and S. Shukla. System level design methodology for SoCs using Multi-Threaded Graphs, *in Proceedings of IEEE International SOC Conference*, 2005 [209].

The author of the dissertation was directly responsible for the the listed publications, from the initiation of ideas on the drawing board to a full fledge contribution to the formal methods community. In the process, the author defined, developed and implemented the solutions for the problems proposed in this thesis. The discussions with co-authors and the comments from peer reviewers have been one of the reason for the success of this work. With the guidance of Dr. Sandeep Shukla, and numerous discussions with Deepak Mathaikutty, various ideas were formulated. The formal underpinning in [198, 215] were developed by Dr. Sandeep Shukla and Dr. Jean Pierre Talpin, whereas the proof sketch and the mapping to SIGNAL was developed by the author. The design transformations formulated in the functional language SML [212, 214, 206] were developed as result of discussion with Deepak Mathaikutty. The latency insensitive protocols were developed from discussions with Dr. David Berner. Michael Kishinevsky and Sava Krstic from the Intel Corporation supervised the author through the verification process of elastic controllers formulated at Intel.

1.6 Organization

This dissertation is organized as follows:

In chapter 2, we discuss related work including asynchronous designs and their classifications followed by a discussion on synchronous design languages. We then discuss latency insensitive protocols and the design flow that leads to latency insensitive designs. We then discuss desynchronization [43, 122] and its relevance to GALS followed by a discussion of GALS and existing works in the area. We conclude the chapter by small discussion on dataflow architectures and time-triggered architectures and how they facilitate a GALS design.

In chapter 3, we provide some background information related to formal methods, model checking, and the formal verification tools we use in this dissertation. We also briefly discuss functional programming as we use SML, a functional language, for simulation based validation.

In chapter 4, we present our the relay station based protocol. We provide the functional description of the protocol as well as its SPIN [130] and SML [20] descriptions. Since, these relay stations have the criteria that they must be placed at strategic points to ensure that the signal propagation of each wire segment is less than the clock period, we present a different protocol based on wires which we call the bridge-based protocol. In this protocol, the relay stations are replaced by additional wires.

In chapter 5, we present our LI validation framework that can be used for validating latency insensitive protocols. Our framework is based on checking latency equivalence between a synchronous design and it latency insensitive counterpart.

In chapter 6, we show formal verification of elastic controllers. This section involves the collaboration work done at Intel during an internship with Michael Kishinevsky and Sava Krstic. We show how the application of elastic protocol [89, 88] for soft IPs satisfies the definition of the elastic machines [148]. We also verify the protocol in our validation framework.

In chapter 7, we present a trace-based formal framework for analyzing designs for transformation to correct-by-construction GALS. Our framework is based on observation of input/output traces of the IPs. We also present a protocol based on barrier synchronization to enable this correct-by-construction transformation.

In chapter 8, we present a formal refinement based framework which takes us from a KPN specification towards a GALS implementation. We consider four different GALS architectures, and provide the formal definitions for each of these. The architectures we consider involve different communication protocols [213, 212]. In the *handshake-based architecture*, the communication between processes occur based on well known four-phase handshake protocol, whereas in the *fifo-based architecture*, the communication protocol borrows ideas from latency insensitive protocols with a fifo interface connecting the processes running on different clocks. The *controller-based architecture* consists of a centralized controller that governs the execution of the processes. Finally, we have the *lookup-based architecture* that avoids the signaling based protocols, and the communication happens based on fast data accesses from a lookup storage which is located on-chip. The architectures we present, consist of processes that are continuous and monotonic, and are also correct-by-construction. We show this by transforming the processes into deterministic (I/O) automata, since they have been shown to be continuous and monotonic [156]. We show behavioral equivalence of the KPN and the refined architecture by ensuring that the behaviors of both are latency equivalent.

In chapter 9, we propose a formulation of polychronous systems [122] using true-concurrency structure [215] of Pomsets [131, 169, 189]. This formulation achieves an unambiguous characterization of both synchrony and asynchrony by exclusively considering the scheduling structure [215, 198] of a Pomset. Instead of considering a tag structure [152] to give a model-specific time-stamp to events, we simply consider a Pomset based model where events are partially ordered. Synchrony and asynchrony are captured by considering the structure of the partial order relation that models the causal relations between events.

In chapter 10, we summarize our main contributions, and conclude by a discussion on future directions.

Chapter 2

Related Work

In this chapter, we discuss existing works and solutions proposed in handling the issues related to IP composition.

2.1 Asynchronous Designs in Industry

Asynchronous circuits have been studied for nearly four decades now, but still have not gained an edge over synchronous circuits [54, 53, 52, 66, 93]. The asynchronous circuits have various advantages over synchronous circuits. We discuss their advantages with respect to performance and power below:

High performance: In asynchronous circuits, the next computation step can start immediately as suppose to synchronous circuits where the next computation starts on the transition of the clock. This leads to potential gain with respect to performance. But, a part of this performance improvement is cancelled out by the overhead of adding additional signals for completion detection. However, the performance for asynchronous circuits may vary on the design. For example, techniques such as *double-rail* encoding can be used to exploit *data-dependent* delays [195, 162], to improve the performance of the circuits. Such techniques have been applied to *carry bypass* adders [234] and *lookahead* adders [180]. Furthermore, at SUN Research, comparison of *micro-pipeline* (asynchronous) and a *shift-register* (synchronous) has been done to analyze the performance gains [217, 172]. A *micro-pipeline* is an elastic FIFO consisting of a cascading structure of latches and controllers that pass data using handshake signaling [195]. It was realized that a carefully designed controller circuit for the micro-pipeline and its corresponding shift-register had almost equal throughput.

Low power: The power in asynchronous circuits is consumed during a transition and based on what parts of the circuit are activated. On the other hand, synchronous circuits are active during the transitions of its clocks, and not all of the transitions are useful. There have been various experiments where asynchronous designs are clearly better than synchronous designs. At Philips Research Laboratories, asynchronous circuits were used for Reed-Solomon error correctors operating at different audio rates [53]. In [52], two different decoder implementations (single-rail and double-rail) were compared with their synchronous counterparts. It was shown that the asynchronous decoders consumed up to five times less power than their synchronous versions. Similar gains with asynchronous designs over their synchronous counterparts were realized in [178, 159, 179, 110, 184]. In synchronous circuits, techniques such as *clock gating* can be used to reduce the power consumption for synchronous circuits, however, it complicates the process of functional validation and timing verification [119, 223].

There are various other advantages and disadvantages of asynchronous circuits, and the reader is directed to [54, 93]. Next, we look at some classes of asynchronous circuits.

2.1.1 Classes of Asynchronous Circuits

Asynchronous circuits can be classified as follows:

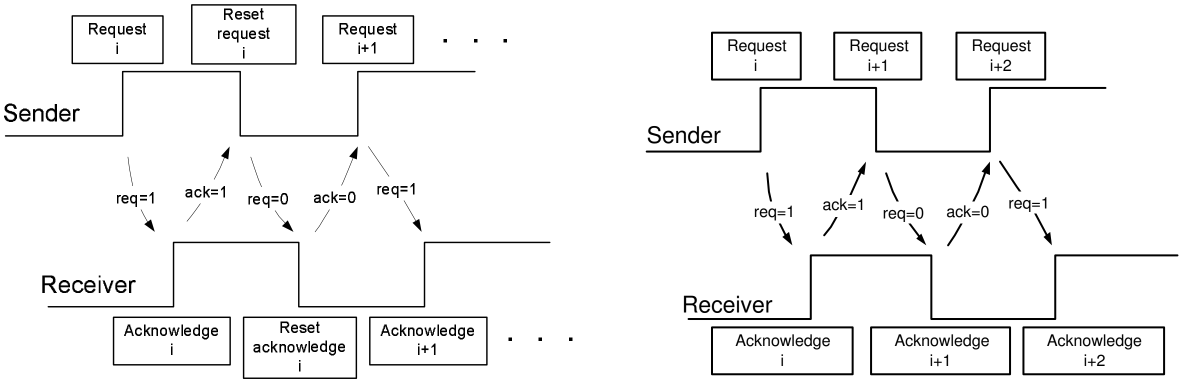
Delay-insensitive (DI) circuits: A DI circuit is designed to operate correctly regardless of the delay on its gates and wires. There can be unbounded delay on the wires and gates, and the circuit would still work correctly [224, 98]. The class of DI circuits is extremely limited and almost no useful DI circuits can be built if one is restricted to a class of simple gates and operators [161, 63]. However, many practical components can be built if complex components, which are built out of several gates, are considered. Some of the simple components of a DI circuit include a wire (one input - one output), toggle (one input - two outputs), C-element or a join (two inputs - one output) and a merge (two input - one output). For more information on DI circuits, the reader is directed to [103]. Using these components, one can build a more complex component such as a *modulo-3* counter [103].

Quasi-delay-insensitive (QDI) circuits: A QDI circuit is delay-insensitive except that it requires *isochronic forks*, where all outgoing branches are assumed to have either the same delay or within some bounded skew [66]. This compromise over DI makes it feasible to build more practical circuits using simple gates and wires [160, 51]. More details of QDI, and its advantages and disadvantages can be found in [161, 50].

Speed-independent (SI) circuits: A SI circuit differs from a DI circuit with regards to the delay on the wires. In SI circuits, the delay on wires is considered to be negligible, however a bounded and positive delay is assumed for gates. More examples of SI can be found in [39, 170].

2.1.2 Controlling Asynchronous Circuits:

Various methods can be implemented to control the asynchronous circuits. One way is to add controlling signals that dictate the start and completion of an operation: a request signal initiates



(a) Four-phase handshake

(b) Two-phase handshake

Figure 2.1: Handshake Protocols

the start of an operation, and when the operation is complete, an acknowledge signal is emitted to indicate the completion of an operation [92]. Another way is to use a delay element, such as an inverter chain, that generates an acknowledge signal based on how long the circuit will take to compute a valid result. Furthermore, special arithmetic circuits can be used to generate acknowledge signals based on carry propagation patterns [132].

Various protocols can also be implemented for controlling asynchronous circuits. Most commonly used are the signaling-based handshake protocols.

Signaling-based Handshake Protocol

The handshake in most asynchronous circuits use signaling protocols that involve requests and acknowledgements [218]. Such type of signaling protocols have been used in various architectures including dataflow computing [28, 92].

Four-phase handshake In a four-phase handshake between a sender and a receiver, the sender sends a request (req=1) for sending a data. Upon receiving this request, the receiver responds by an acknowledge signal (ack=1) denoting that it has received the request. These signals are then reset to ‘0’ to allow for next transaction. Figure 2.1(a) illustrates this behavior.

Two-phase handshake In a two-phase handshake, the sender and receiver communicate by sending the same data as in the four-phase handshake, however, the difference here is that the every change detected in the request and acknowledge signals is considered as a new trigger. Figure 2.1(b) illustrates this behavior.

It can be realized that a two-phase handshake works better with respect to power and performance, since every transition represents a meaningful event. In the case of four-phase handshake, the request and acknowledge signals need to be reset for the next transaction. However, since the two-phase handshake requires more complex logic for detecting change in current and previous values, power saved by this reduced control transmissions is offset (as seen in the case of ARM processor [108, 109]).

Various encoding schemes such as dual-rail encodings can also be used in the form of communication protocols for asynchronous circuits [94, 227].

2.2 Synchronous Design Languages

VHDL [205, 87], Verilog [222], and SystemVerilog [219] are the most popular synchronous design languages that are primarily used for hardware description and modeling. These languages follow the discrete-event model of computation (MoC) [152, 151], and allow concurrent processes to be described procedurally. VHDL, derived from Ada [64] programming language, is a richly-typed language, whereas Verilog is a limited-typed language allowing a four-value bit vectors and arrays for modeling memory [104]. SystemVerilog extends Verilog by adding a rich user-defined type system.

SystemC [13] language is a set of C++ classes for system modeling that builds from Verilog and VHDL-like modules. SystemC also uses a discrete-event model of computation, but has a flexibility of operating with a general-purpose programming language. The extensions of SystemC with different models of computations have been presented in [183]. SpecC [113] is an extension to C and is aimed towards system-level design. Both SystemC and SpecC allow description of software as well as hardware within a single unified framework.

Lustre [123] is a declarative synchronous dataflow language mainly used for describing reactive systems. The objects are represented by flows which are an infinite sequence of values. These flows are defined by their equations. Lustre consists of the usual operators: boolean, arithmetic, comparison and conditional. It also has two temporal operators for describing sequential functions: pre and $\rightarrow$. For a flow x, $pre(x)$ denotes the previous value of x. At the very first step, the value of $pre(x)$ is nil. On the other hand, for any two flows x, y, $x \rightarrow y$ denotes that the flow in the first step is equal to x, followed by y thereafter. The Lustre program has a master clock which dictates the behavior of the program. All the flows of the Lustre program are associated with the same clock.

While Lustre is declarative and mainly focuses on dataflow aspects, Esterel [56] is an imperative language and is well-suited for describing control. An Esterel program consists of a collection of nested, concurrently executing threads that are described imperatively. The threads communicate with each other using signals. The input and output signals can be either pure signals (carrying presence or absence) or valued signals. The communication between threads occurs through synchronous broadcast. *Causality analysis* is done to ensure that the Esterel program is deadlock-free [44]. Esterel was initially designed for single clock hardware and software designs, however, its extensions to handle multiple clock zones were recently proposed [55]. In multiclock Esterel, statements driven by different clocks communicate through two special devices called the sampler and reclocker [55].

SIGNAL [47, 121] is a multiclock synchronous declarative language. Similar to Lustre, the objects are represented as flows which are an infinite sequence of values. SIGNAL program does not consist of a global clock, which provides declaration of multiple clocks. SIGNAL is more expressive than Lustre, however, its clock calculus is more complex. In SIGNAL, each signal is associated with a clock, and a type (boolean, integer, real,..) with the domain augmented with a special symbol $\perp$ that represents absence of a value. SIGNAL consists of small number of constructs which we discuss in Chapter 9.

SystemJ [120] is a design language proposed for modeling globally asynchronous locally synchronous designs. It is an extension of Java with synchronous reactive features of Esterel [56] and CSP-like asynchronous constructs [129]. Each local synchronous island consists of composition of concurrent reactions that are synchronous. These islands communicate with each other using CSP-like rendezvous communication. Thus SystemJ follows the GALS paradigm where synchronous blocks communicate asynchronously.

2.3 Latency Insensitive Designs

Latency insensitive protocols (LIPs) have been proposed as a viable means to connect synchronous IP blocks via long interconnects in a SoC. As discussed in the previous chapter, the reason why one needs to implement LIPs on long interconnects stems from the fact that with the increasing clock frequencies, the signal delay on some interconnects exceeds the clock period. Therefore, the signal takes longer than a single clock period to propagate between IPs. LIPs ensure proper communication over these long interconnects. To render a design to be latency insensitive design, a design flow based on synchronous composition and stepwise refinement can easily be conceived.

2.3.1 Design Flow for LI Refinement

In this section, we describe a transformation procedure to refine a synchronous system to an LI system. Figure 2.2 illustrates this flow diagram.

The steps to LIP refinement are as follows:

1. We start with a collection of synchronously communicating components. These components can represent custom-made modules or IP cores.

2. Floor planning and interconnection routing are done to check for long interconnects. If all communication can be done in a single clock cycle, there is no need for LIP refinement.

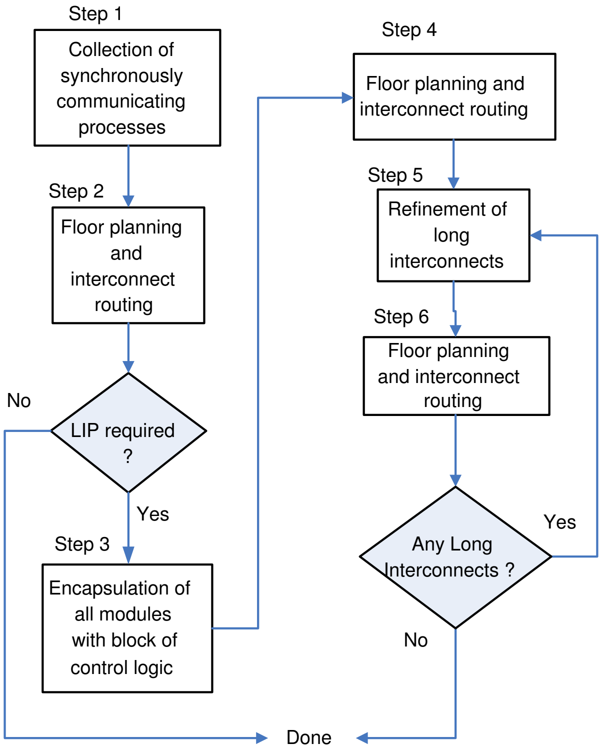

Step 1
Collection of synchronously communicating processes
Step 2
Floor planning and interconnect routing
No
LIP required ?
Yes
Step 3
Encapsulation of all modules with block of control logic
Step 4
Floor planning and interconnect routing
Step 5
Refinement of long interconnects
Step 6
Floor planning and interconnect routing
Yes
Any Long Interconnects ?
No
Done

Figure 2.2: Refinement steps to LI implementation

3. If long interconnects are present then all modules are encapsulated with a block of control logic. This encapsulation includes logic that controls the flow of the events, buffers, control stations, repeater stations etc. to enable correct transmission of data.

4. Estimation using floor planning and interconnect routing is done again, this time with the encapsulated processes to relocate and evaluate the delays on the long interconnects.

5. After finding the delays on the long interconnects, the designer can segment those long interconnects with additional processes containing buffers, latches, forwarding stations, etc to ensure that data is properly communicated through the long interconnects. Depending on the delay of the interconnect, the events can be compared from the point they are placed on the signal to the point they leave the signal.

6. Floor planning and interconnect routing is done again to ensure that no long interconnects exist in the system.

We now look at the different LIPs proposed in the literature.

2.3.2 Latency Insensitive Protocols

Several approaches have been proposed on how to create LI designs. These approaches require adding *valid* and *stall* signals to the existing data wires, adding a wrapper to each module for communication using these signals, and finally adding some mechanism to handle communication over long interconnects.

Carloni et al. proposed a "correct-by-construction" methodology to design latency insensitive systems for single clock SoCs [68, 70]. In their approach, all modules (called *pearls*) are encapsulated in a wrapper to form a "shell". Their encapsulation makes the shells to be *patient processes* whose functionality depends only on the order of the events of its signals, and not on their exact timings [68]. The wrapper encapsulation does not require modifications of the internal logic of the *pearl*, however it involves composing each module with an *equalizer*, a *stall signal generator*, and an *extended relay station*. The role of the equalizer is to align the inputs provided to the *pearl*, and for each input signal it has two storage elements that act like a queue. Based on the inputs realized by the equalizer, the stall signal generator places appropriate stall values on its stall signals. The *extended relay station* is placed on the output of the *pearl* to store the values in case the shell at the output stalls. *Relay-stations* (also patient processes) are placed along the long interconnects. They act like pipeline blocks to store and forward data, and contain at least two registers and control logic. The shell or the relay station is stalled whenever the stall signal is high for two consecutive clock cycles. They also provide the theory supporting the correctness claims [70]. However, in their recent paper [153], they proposed another optimized protocol for latency-insensitive design, where a single stall signal would stall the process. They provide the details of their protocol and its verification in [153]. However, their verification methodology is closely related and based on our verification methodology for checking correctness of the latency insensitive protocols.

Casu and Macchiarulo show how they reduce chip area compared to Carloni's approach [72]. They use a scheduling algorithm for the functional block activation, and substitute relay stations with simple flip-flops. One main disadvantage of their approach is that the schedule has to be computed a priori and depends on the computation in the process. If any change is made in any process, it may result in the change of the flow of tokens and also may result in inconsistency with the current scheduling algorithm. In this case, the schedule has to be recalculated, which is expensive.

Boucaron et. al [200, 59] formalize the descriptions of the relay stations and the shell wrappers for latency insensitive designs, and provide their respective correctness properties. The correctness properties focus on the control flow aspects of tokens through the components. They also provide a solution for balancing the latencies on the paths in a latency insensitive design. They analyze the design as a weighted event/marked graph, where the computation nodes are shown as vertices and the latencies are marked on the arcs. The relay stations are inserted based on the latencies. However, this may still not make the graph balanced with respect to the number of latencies on the paths from the source to the destination nodes. Now to ensure that the graph is equalized based on the latencies, they insert new elements called fractional elements on the faster arc to slow them down. These fractional registers are mainly used in graphs with nodes that have loops (strongly connected) case. These fractional registers are statically scheduled based on the flow of tokens through the path. This idea is based on earlier work by Casu and Macchiarulo [72], where a static schedule is computed for firing each computational component. The authors also provide solution for the case where the graph is a directed acyclic graph (DAG). In [60], they also provide an extension where the components can be considered to have independent clocks.

Singh and Theobald [201] generalize the LIPs for complex network topologies, where they propose extensions to point-to-point communication channels to include *forks* and *splits*. Such specialized blocks can be arbitrary connected to form a rich set of communication topologies. They also extend LIPs for GALS systems, where different components can run on their independent clocks. In their approach all input and output signals are controlled by complex FSMs implemented in the wrapper. [29, 30] provides an extension to this work by presenting detailed architecture and wrapper synthesis techniques. They provide an automated tool that accepts interface specification in component wrapper language (CWL), a high level specification language [33], and produces gate-level implementation of the wrapper circuitry.

Cortadella et al. [89, 88] presents another LIP called *synchronous elastic flow* (SELF) which is used to realize a latency insensitive or as they call an *elastic* design. An elastic design is made up of elastic modules that are insensitive to latencies on the interconnects. Their protocol involves transforming all the data wires to channels with data, valid and the stop signals, and replacing all flip-flops with *elastic buffers* (EBs). An EB consists of two transparent latches with opposite polarity that store data based on the control logic. This control logic also communicates with the channels and governs the flow of tokens through the EB. Figure 2.3 illustrates the SELF protocol, where each channel can be in three states: $idle$, $transfer$, and $retry$. The transition to different states occur based on the values of valid (v) and stall (s) signals of the channel. They provide a complete synthesizable description of the *elastic buffers*.

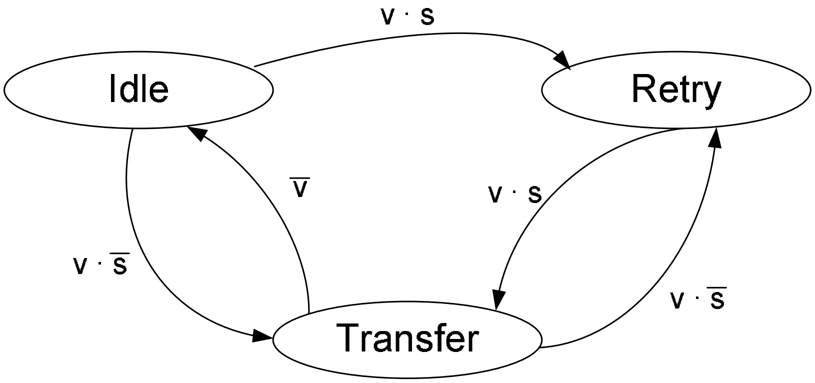


Figure 2.3: Elastic Channel Protocol

This protocol primarily considers a white box model targeting pipelining in microprocessor designs, however, it can also be applied to black box models. They have also developed the theory of *elastic machines* that supports their protocol and provides properties for an *elastic machine* [148]. A design is elastic if the following three properties hold: persistence, backward liveness and forward liveness. The persistence property states that an elastic component must remain ready until the transfer takes place. The liveness properties are expressed in terms of transfer counts on the input and output channels, where forward liveness ensures that transfers occur on output channels, and backward liveness ensures that transfers occur on input channels. These properties are explained in detail in [148].

Stevens et. al [230, 111] modify the SELF protocol and propose a new interleaving protocol shown in Figure 2.4. This protocol also has the same performance and design benefits of the original protocol but it is targeted towards network communication and desynchronization. The interleaving protocol has the advantage that it can ensure direct communication with asynchronous channel where request and acknowledge signals switch in alternating phases. This protocol supports direct communication between the clocked and asynchronous logic, and can be directly mapped and synthesized into either clock or asynchronous network fabrics [230].

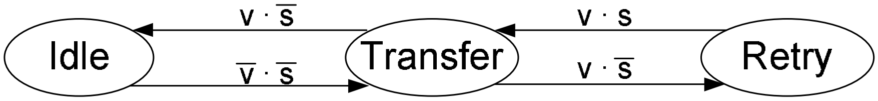


Figure 2.4: Phased Elastic Channel Protocol

2.4 Desynchronization

Desynchronization of synchronous language specification was studied for distributed deployment for synchronous specification, mainly to analyze the asynchrony aspects of GALS. The theory of *desynchronization* was developed and its correctness was formally proven in [41]. Also, various characterizations of synchronous systems have been realized to achieve a GALS implementation. These are specifically the notions of *endochrony* and *isochrony* [42]. Benviniste et. al [41] proposed that a synchronous system with the property of *endo-isochrony* can be implemented as a semantically equivalent GALS architecture without the need for any protocol. An *endo-isochronous* system has the property that each component is *endochronous* and the communication between any two of its components is *isochronous*. A synchronous component is said to be *endochronous* when the presence or absence of each variable can be inferred incrementally given the values of the state variables and current inputs. The notion of ‘endochrony’ is used to decide the feasibility of single clock implementation of a specification, or of a distributed implementation over asynchronous channels. However, the mathematical formalisms used to define such notions are highly abstract, and are difficult to comprehend intuitively for a given program. *Isochrony* is a condition that mainly applies on common variables of two synchronous components where they must agree on the values which are assigned to it at each time instant. Hence, there must not exist a pair of reactions that is contradictory (i.e. both have different values on some common variables). [186, 187] extends the notion of endochrony and isochrony to make these properties appropriate for composition. They introduce the notions of weak-endochrony and weak-isochrony, which they claim to be a criterion for guaranteeing correct distribution of synchronous specifications [188].

Desynchronization techniques have also been studied for transforming synchronous designs to asynchronous designs. In [61], a methodology for converting synchronous circuits to low power asynchronous circuits is described. A doubly latched asynchronous pipeline architecture [141] is employed that can adapt its effective operating frequency to the supply voltage, facilitating flexible and efficient power management [61]. [90] shows different protocols for desynchronization and formally proves their correctness. Some involve replacing the flip-flops with the latch based controllers with handshaking circuitry [141, 226], while others attempt to replace logic gates with sequential handshaking asynchronous circuits with encoding signals with delay-insensitive code [154]. In [91], various optimization techniques are employed to reduce the overhead of desynchronization. These techniques are based on clustering and temporal analysis of desynchronized designs.

2.5 Globally Asynchronous Locally Synchronous

A globally asynchronous locally synchronous (GALS) design consists of smaller components with independent clocks. These components are usually developed using standard CAD tools and design flow, and the communication between them happens asynchronously. In this section, we discuss different GALS techniques and architectures proposed in the literature.

One of the well known methodologies for GALS design is to use pausible clocks for controlling the synchronization aspect of inter-module communications [232, 233, 177, 175, 139, 173, 181, 228, 202] to avoid metastability. In this approach, the synchronous components are encapsulated within an asynchronous wrapper. Each locally synchronous block uses a ring oscillator to generate the clock. The clocks of the transmitter and receiver are stopped for data transfer between these asynchronous wrapper. The communication between the asynchronous wrappers is done though handshaking.

Using FIFO interfaces is another way of designing GALS. FIFO buffers are placed between synchronous components. A design with such interfaces cannot only handle long interconnect delays, but also provide an acceptable throughput between components [78]. Furthermore, these FIFOs help in handling synchronization between components. In [79], mixed timing FIFOs were introduced for communication between synchronous-synchronous (multi-clock domains), synchronous-asynchronous and asynchronous-synchronous domains. Furthermore, mixed timing relay-stations were also introduced for long interconnects. Their design uses a two-flop synchronizer for handling metastability. They also present designs for multi-clock relay stations that can be used for latency insensitive designs with multiple clocks. In [75, 76], source-synchronous communication was proposed for different clock domains. In [77], abstract timing diagrams were used to analyze the interfaces between the synchronous domains, whereas in [99], signal transition graphs [81] where used.

A modified-two phase asynchronous interface wrapper for communication between two locally synchronous modules is presented in [173]. They also propose a FIFO buffering mechanism to enhance performance between inter-module communications. In [181], a four-phase asynchronous wrapper is presented that handles multiple ports. A number of GALS interconnect structures are analyzed in [228], involving ring topologies and packet-based communication.

Loosely synchronous interfaces can also be employed for designs where the frequency relationship between communicating blocks is well defined [76, 75, 167, 77]. In such cases, a designer may relax constraints about clock skew, and take advantage of having known clock frequencies. A system can be classified into one of the following clock frequency classes [74]: (i) Exactly matched clock frequencies (EMCF), (ii) Rationally related clock frequencies (RRCF), (iii) Nearly matched clock frequencies (NMCF) and (iv) Arbitrary clock frequencies (ACF). There is also an orthogonal classification of systems presented in [115, 168] as synchronous, plesiochronous, mesosynchronous, multi-synchronous. A system classified as EMCF has all its components run on the same clock. The IPs are designed to assume that informative values are present on every signal at the arrival of the clock. A value is *informative* if it is of relevance to a system. In real hardware, every block reads and writes informative values on their corresponding input and output signals at every cycle. If these values are to be ignored or older values are to be reused, techniques such as clock gating can be used [37, 62]. Next, we have systems classified as RRCF, where the components run on different but related clocks. In such systems, the clocks of all the components are related at the root of the *clock tree*. Let $C = \{c_1, c_2, \ldots\ c_k\}$ be the set of all the clocks for a system, and $w(c_i)$ be the frequency of the clock. An edge e is defined as $e = \{(c_i, c_j) : w(c_i) | w(c_j)\}^1$, where | is an acyclic relation and $c_i \neq c_j$. If $G = (C, E)$ is a directed graph, where E is the set of all edges, then we call it a *clock tree*. The frequency of the clock at the root of such a tree is obviously the multiple of every clock's frequency in the system. A *clock tree* of a system expresses the relationship of all its clocks. The sub-clocks are created generally by using frequency dividers/clock dividers. The clocks have sampling points at which the new events occur. We refer to these sampling points as *clock ticks*. Consider a producer and a consumer, where the clock frequency of the producer is twice that of the consumer. Now, the producer can produce two values in two clock ticks before the consumer can read them in its one clock tick. Next, we have systems with NMCF, also referred to as plesiochronous, where the frequencies are close but different. For example, in the design of network routers where each line card receives a bit stream. These streams are generated from different clock sources which are closely related in terms of their frequencies [74, 145]. The systems classified as ACF consist of clocks that are completely independent, and arbitrary. Such systems may have its components driven by some external clock sources. There are other classifications of systems such as mesosynchronous and multi-synchronous, where systems have same clock frequencies, however differ in the phases [115, 168].

In [99], authors present a possible scenario that can lead to a metastable state due to clock tree delays in a GALS wrapper with stoppable clocks. Hence, they propose a modified wrapper protocol to correct this problem. Furthermore, they propose a locally delayed latching (LDL) scheme for inter-module communication in GALS design. This LDL scheme is insensitive to clock tree delays,

[1] | denotes divides.

and does not employ stoppable clocks. In [167], delay augmented netcharts are presented as a formalism for representing communication protocols. They use flow control protocols to optimize synchronization in rationally clocked GALS systems.

In [174], Mousavi et al. propose a formal framework for validating a GALS design in a synchronous framework of SIGNAL. They theoretically show equivalence of a design composed of two asynchronously composed synchronous components to a fully synchronous multi-clock model preserving its behavioral equivalence.

2.6 Dataflow Architectures

The architectures for dataflow computing were developed in the 80s and 90s, and we use some of the ideas presented in these architectures which include the static and dynamic dataflow architectures [95, 36].

In the static dataflow architectures, the information specifying the instruction to be executed, operand values, and destination address fields [34] are encoded into specific templates called *activity templates* [96]. These *activity templates* are then decoded by each node (referred as processing elements), where each node consists of various units that enable retrieving and storing of the information based on the presence of the appropriate operands. The operand slots in these *activity templates* consist of present bits that indicate the presence of the data. If sufficient data is present for an operation, then the instruction is executed, and its result is forwarded to the successor nodes. The architecture consists of handshake signals to pass the result to the successor nodes. Also, the instructions are executed in a FIFO manner with at most one token (valid data) on each arc. No more than one token per arc is permitted in the architecture. The reader is referred to [35] for more details about static dataflow architecture. We also explore the idea of using message structures containing operands with present bits indicating valid data which is similar to activity templates [34].

Dynamic architectures were created to increase parallelism which were rather limited in static architectures due to handshakes [35, 34]. The dynamic architectures used the concepts of tagged-tokens, where tokens consisted of a three part tag which were used to identify the context of the operation, the node, and iteration number for loop iterations. A “waiting-matching unit” was used for collecting tokens destined for binary operators, and pairing them with the corresponding matching tokens. If a matching token was found, the operation was enabled. Again, the reader is referred to [35, 34] for details on dynamic dataflow architectures. We use the idea of “waiting-matching unit” for GALS as well, where the data is stored and retrieved from a on-chip lookup store, which is a small on-chip storage.

2.7 Time Triggered Architectures

Time triggered architecture (TTA) was proposed by Kopetz for the specification for the design on a distributed platform [143]. The architecture employed the use of global time for communication protocols to perform error detection and guarantee timeliness of real-time applications. A two-phase design methodology was employed by TTA which included architecture design and component design. In the architecture design phase, the interaction between distributed components and the interfaces of the components are completely specified in the value and temporal domain [144]. The component design is then developed using these interfaces as constraints. The TTA uses a spare-time model where the continuum of time is partitioned into an infinite sequence of alternating durations of activity and silence. The duration of the activity is assumed to be larger than the precision of the clock synchronization [144]. Event that occur on the same global clock tick occur simultaneously, whereas events that occur during different activity interval durations, are ordered based on their global timestamp. However, the problem with time triggered architectures is that it requires distributed platform to have clocks that are perfectly synchronized, which may be problematic. [45, 46] presents an implementation architecture which is a relaxation of TTA called loosely time triggered architecture (LTTA), where the writes and the reads from the communication medium is independently triggered by quasi-periodic clocks which are non synchronized.

Chapter 3

Background

In this chapter, we discuss the relevant definitions pertaining to our work. We discuss formal methods along with some examples. We then elaborate on model checking. We further describe some frameworks and tools used for model checking, and briefly discuss functional programming.

3.1 Formal Methods

Formal methods include ways of describing the behavior of systems mathematically and reasoning about them to prove correctness and analyze their behavior and performance. Formal methods are needed in many fields of software and hardware systems, and more significantly used for the safety critical systems to ensure that they behave in a way they are supposed to behave. Errors in such systems may result in damages in various aspects. Various techniques of formal analysis have been developed, each having its own advantages and disadvantages.

One such method of formal analysis is model checking [85, 190]. Model checking by using temporal logic was developed by Clarke and Emerson [85, 84] as well as by Queille and Sifakis [190] in the early 1980s. Later on, many model checking tools such as SPIN [130], SMV [166], UPPAAL [24], etc were developed. Model checking has been one of the popular formal methods used in the industry and we have used some of these model checking tools to illustrate our methodology. We discuss more about the tools and methods used in our work in the later part of this chapter. There are other techniques for formal analysis such as theorem proving [216, 23, 7, 26], equivalence checking [197, 158], etc. Theorem proving systems are computer programs capable of proving theorems and proofs by formulations of the problem as axioms, hypotheses and a conjectures whereas equivalence checking involves checking for equivalence between two systems which may be described using HDLs or as gate-level circuits, or in any other suitable formalism. However, in this thesis, we use model checking approaches, and will not focus on theorem proving or equivalence checking.

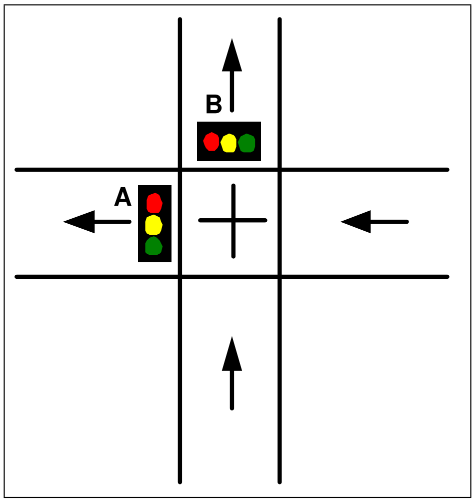


Figure 3.1: Traffic light controller

Formal methods employ various modeling frameworks [101] for specifying and describing various systems. For a system to be correctly specified for its intended behavior, the correct framework has to be chosen. Various modeling frameworks such as finite state automata [130], petrinets [176], stochastic modeling framework [158], etc. exists in the industry today for analyzing systems.

To illustrate the applications of formal methods, we will consider an example of a traffic light controller shown in Figure 3.1, which shows an intersection with two signals: A and B. The finite state machine representation of the traffic light controller is shown in Figure 3.2 which consists of four states and a set of transitions. Each state has specific values for signals A and B. The signals can have three possible values: red, yellow and green. The set of possible transitions that A and B can take is described in the automaton in Figure 3.2. The formal definition of this system is as follows:

Let $X = \{A, B\}$ be the set of state variables.
Let $W = \{red, yellow, green\}$ be the domain of A, B
then, $P = \{(x = w) \mid x \in X, w \in W\}$ is the set of primitive predicates.
Let $S = \{s_1, s_2, s_3, s_4\}$ be the set of states. Note that $S \subseteq X \times W$. $X \times W$ has some combinations where (x, w) are not possible.
then, $P_{s_i} \in P$ is the set of predicates true in s_i for example $P_{s_0} = \{A = green, B = red\}$ in state s_1.
Let $L = \{(x := w) \mid x \in X, w \in W\}$ be the set of action labels.
Let L_{t_k} be the set of actions taken on t_k transition. For example, L_{t_1} represents the action taken on t_1 which is $A := yellow$.
Let $T = \{(s_0, L_{t_0}, s_0), (s_0, L_{t_1}, s_1), (s_1, L_{t_2}, s_2), (s_2, L_{t_5}, s_2), (s_2, L_{t_4}, s_3), (s_3, L_{t_3}, s_0)\}$ be the set of transitions where $T \subseteq (S \times L \times S)$

This formalism helps in clearly stating the intended behavior of the traffic light controller. From

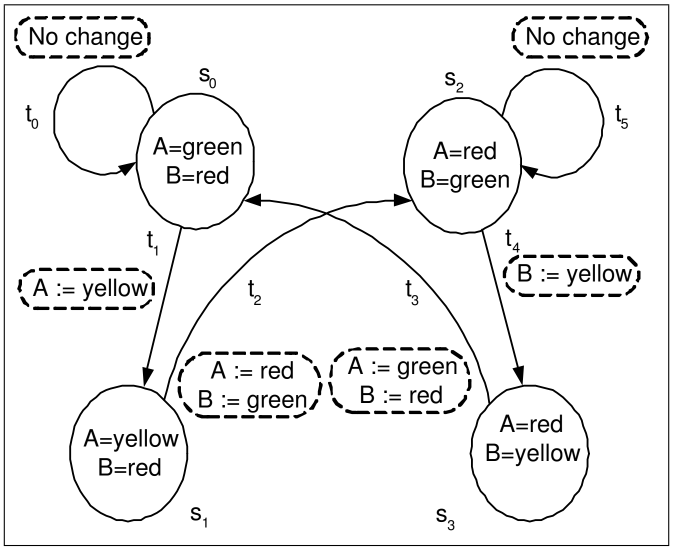


Figure 3.2: Finite State Machine of a traffic light controller

the formal description and the finite state machine shown in Figure 3.2, it can be clearly seen that the state where both signals are green does not occur which is an erroneous behavior.

Next, we define what model checking is, how it is done and also discuss what tools we use to experiment our methodology.

3.2 Model Checking

Model Checking [86] is an automated technique for verifying formal models of systems. Applying model checking to a design consists of several tasks that include modeling, specification and verification. The modeling aspect involves converting a design into a formal language representation that can be accepted by the model checking tool. This procedure may require abstracting away irrelevant details of the system that are not relevant for the actual intended behavior. Abstraction helps to reduce the model size of the system, which prevents the problem of *state space explosion* which we discuss later in this section. Specification of the properties require stating the properties in some logical formalism which can be expressed in many different ways such as temporal logic properties [185], finite state machines [130], first order logics [83], etc. The model should be designed such that it satisfies the required properties. Once we have the specified property and the formal model of the system, we then verify the properties against the design. The model checking system may either terminate with a successful verification of the system or will give a counter example in the form of a trace indicating that the system failed to satisfy the given property. The user can then locate and fix the error, which either may be in the modeled design or may have occurred due to incorrect specification of the system that is being modeled. The problem of model checking can be described as follows:

Let M be the model and let $S = \{s_0, s_1, ..., s_i\}$ be the set of states in the model.
Let φ be the formulae to be checked.
then,

model checking problem is whether $M, s_k \models \varphi$, which means whether the formulae φ is satisfied in s_k for $k = 0...i$.

Referring to the example of the traffic light controller discussed in section 3.1, let us consider a safety property that states whenever signal A is green, then signal B is red. We have the property φ being $[](A = green \rightarrow B = red)$. This property will hold true for the model at all states since whenever, signal A is green which is *true* at state s_0, then signal B is red. Apart from state s_0, signal A is never green. Now, let us consider another property that states whenever signal A is red, signal B is green. We have the corresponding φ as $[](A = red \rightarrow B = green)$. This property will not be satisfied since in state s_3, signal A is red but signal B is yellow and not green. Therefore, the model checker will generate a trace showing that at state s_3, the property does not hold.

The main challenge in model checking is the problem of state space explosion [166]. This problem occurs in large and complex systems with many components that interact with each other or in systems with data structures that can assume many different values. Some of the techniques can be applied to alleviate the problem of state space explosion that include abstraction [97], symbolic representation [118], partial order reduction [147], symmetry [86], etc. These techniques help reduce the model size of the system thereby easing the process of verification.

3.2.1 Temporal Logic

Temporal logic [86] is a formalism that allows us to describe and reason about causal as well as temporal relations of properties. The analysis of systems by using temporal logic was initially used by Amir Pnueli in 1977 [185]. Temporal logic is used for specifying order of events in time without introducing the time explicitly. Operators such as $Eventually$, $Always$ or $Never$ are used to describe events and their temporal aspects in a system.

We talk about some of the common temporal operators in the next section. There are many advantages of using temporal logic to specify systems. One of the main advantages is that it has a well-defined semantics. It is independent of the modeling language and it has enough expressiveness. Temporal logic can be classified into *linear temporal logic* (LTL) and *computation tree logic* with the distinction being how they handle the branching in the underlying computational tree where a computational tree shows all possible execution paths starting from the initial state. LTL consists of a linear time structure and consists of temporal operators to reason about the behavior along the path, whereas CTL consists of a branching tree structure and consists of temporal operators to reason about behaviors among multiple paths [166]. Referring to the traffic light example in section 3.1, let us consider a property that states if signal B is yellow, it will eventually become red. This property in LTL is specified as $[](B = yellow \rightarrow <> (B = red))$. On the other hand, in CTL, the property can be specified as $AG(B = yellow \rightarrow AF\ (B = red))$.

In our thesis we mainly focus on linear time temporal logic (LTL) which we define in the following section.

3.2.2 Linear Time Temporal Logic

In linear time temporal logic(LTL), it is possible to define properties for all reachable states in a single computational path. We show some of the commonly used syntax and semantics of LTL with the example of the traffic light controller discussed in section 3.1. For a complete formal treatment of LTL, one should refer to [86]. Revisiting the example, we have two signals A and B. Both signals can have three different values: red, yellow and green. We show some of the properties based on the example to illustrate the temporal operators below:

Property 1. *Always - It is denoted by* $[]$ *symbol.*
Property: Always both signals A and B cannot be green at the same time.

LTL property: $[](\neg(A = green\ \&\&\ B = green))$. *The property captures the notion that at any time, signals A and B will never be green simultaneously. The property is considered to be a* safety *property.*

Property 2. *Eventually - It is denoted by* $<>$ *symbol.*
Property: When signal A is yellow, it will eventually change to red.

LTL property: $[](A = yellow \rightarrow \quad <> (A = red))$. *The property captures the notion that whenever A is equal to yellow, it will eventually become red.*

Property 3. *Next - It is denoted by* X *symbol.*
Property: If the signal A is yellow, it will change to red in the next state.

LTL property: $[](A = yellow \rightarrow X\ (A = red))$. *The property captures the notion that whenever signal A is yellow, in the next state, it will change to red.*

Property 4. *Until - It is denoted by* U *symbol.*
Property: The signal B will not change to green until A changes to red.

LTL property: $[]((B = \neg green)\ U\ (A = red))$. *The property captures the notion that always B will not change to green until A is red.*

The properties of the example were not difficult to express but at times, temporal properties are not always easy to write [102]. Therefore, possibilities of errors exist if a user wants to express the behaviors in LTL. A simple way to check if the intended behavior that the user is trying to specify is correctly expressed in LTL is to convert the LTL property into a finite state machine [130]. Finite state machine is a framework for formal modeling and provides an ease of visualizing the expressed behavior. There are tools that automatically convert LTL properties to finite state machines [10, 150, 130].

3.3 SPIN

SPIN [130] is a model checker used extensively for verification of distributed systems including control algorithms, data communication protocols, etc [130]. It was developed at BELL Labs [3] by Gerard Holzmann. It was recognized by ACM (the Association for Computing Machinery) with Software System Award [16]. The specification language used by SPIN is a high level language used to specify system descriptions and is called PROcess MEta LAnguage (PROMELA). SPIN is used to trace logical design errors of various system designs and checks the logical consistency of the specification. SPIN avoids constructing a global state graph as a prerequisite for verification; instead, it works on-the-fly, which means that the states are created during the verification process. Its basic building blocks include asynchronous processes, message channels, synchronizing statements, and structured data. Once the model is built, the user can simulate it with the built-in simulator that supports random, interactive as well as guided simulation schemes. SPIN can be used in three different modes: editor, simulator and verifier, which are integrated into a user friendly GUI. Components are modeled as processes and data is exchanged using shared variables and/or channels. SPIN targets efficient software verification primarily asynchronous systems, however, we use it for verification of synchronous systems. SPIN friendly nature of manipulating arrays allows for parameterization of the components based on number of inputs and outputs. The properties are specified as LTL and assertions within code. Also various optimization techniques such as partial order reduction [80], statement merging [194], hashing [229], etc, make SPIN faster on certain classes of verification problems.

SPIN reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes [18]. It verifies LTL properties by considering LTL properties as never claims in the form of Büchi automata [130, 10, 150]. It then verifies the model against the Büchi automata. If the property is not satisfied, it generates a counter example in the form of a trace file. This trace file can be loaded into the simulator provided by the tool and checked for the errors in the model. For some properties that are not expressible by LTL formulae, it also considers verifying an automata in the form of a never claim against the model. For more information on SPIN, refer to [18].

We use these basic blocks to write synchronous models. The communication is done through shared global variables. Since the processes run asynchronously in SPIN, we synchronize the execution of all processes with a *clock controller* in order to make our model behave synchronously.

3.3.1 PROMELA

PROMELA is an acronym for *PROcess MEta-LAnguage* and it is intended to make easy abstractions of system designs. It is not an implementation language, but a specification language. Specification in PROMELA consists of processes, message channels and variables. The processes are global objects running in parallel unless specified. Message channels can be declared either globally or locally within a process and are used to pass data from one channel to another. Message channels are *rendezvous* [128] in PROMELA meaning that a handshake occurs for send and receive operations in different processes. Variables can also be declared either globally or locally within the processes. Depending on the state of the system, any statement in PROMELA is either executable or blocked.

```
/* The code illustrates a blocking instance. The execution will block at while until the value of
a is equal to value of b where a and b are variables. Once they become equal, then the value of
c will be added to a*/
while (a!=b) {
    skip;
}
a = a + c;
```

For more reference on the PROMELA syntax, refer to [130]. SPIN is mostly used for verification of concurrent software systems rather than descriptions of hardware circuits. On the other hand, SMV [166] which we discuss next is used widely for hardware verification systems.

3.4 NuSMV

NuSMV [82] is a symbolic model checker used for verification of synchronous as well as asynchronous finite state systems. It is a reimplementation and an extension of the SMV model checker [166]. The NuSMV model checker allows for writing of complex components in a modular way, where each module can be instantiated multiple times. It allows several forms of specification, including the temporal logics CTL and LTL, finite automata, embedded assertions, and refinement specifications. This provides the user with a comprehendible description of the components of a large design, and also helps in re-use of modules. Various advanced features of NuSMV allows for simulation as well as verification of larger systems [82]. The model checker combines binary decision diagram (BBD)-based model checking [166] with SAT-based model checking [165, 57] along with a reduced boolean circuit (RBC)-based bounded model checker [27].

3.5 Vacuity check

Vacuity can indicate a serious flaw either in the model or in the property. Antecedent failure, where a formulae containing an implication proves true because the antecedent or the precondition of the implication is never satisfied, is one of the reasons that a property can be vacuously true. Revisiting our example from the section 3.1, consider the following scenario:

Property: When signal A is green and signal B is yellow then in the next state B will change to green.
LTL property: $[]((A = green \;\&\&\; B = yellow) \rightarrow X\ B = green)$.

The scenario mentioned above is wrong and should never happen. However, since the antecedent of the property is always false, the property will verify to be true which is incorrect. Therefore, it is important to check that at some point of time, the antecedent becomes true. One of the ways to verify that can be to check if the negation of the antecedent is never true [38]. For example, the LTL property: $[]\neg(A = green \;\&\&\; B = yellow)$ will be satisfied implying that the antecedent condition never occurs. Instead, the model checker should generate a trace for the property if the antecedent was true at any point, thus ensuring that the property is not vacuously true. Since, we use Cadence SMV as one of the model checking tools, we can also specify the property in branching time logic. For example, the vacuity check for above mentioned property in CTL can be specified as: $EF\ (A = green \;\&\&\; B = yellow)$ which states that there will exist a path in the system which will have signal A as green and signal B as yellow. This check is also referred to as *trigger check*.

3.6 Functional Programming

Functional programming is seen to be highly relevant to the understanding of reactive and interactive systems. A computation is expressed as a function and its interaction with the outside world is modeled as inputs given to the function. Functional languages provide a clean and simple semantic model, which performs all computation by function application, thereby providing a more abstract notation to express computation.

We use Standard ML (SML) [171] for simulation-based validation of designs. SML offers an excellent ratio of expressiveness to language complexity, and provides competitive efficiency. SML manages to combine safety, security, and robustness with a great deal of flexibility because of its type and module system. Other features of the SML based framework include:

- SML provides good expressiveness with its ability to treat functions as first-class values, and its usage of higher-order functions. The availability of imperative constructs provide great expressive power within a simple and uniform conceptual framework.

- SML provides a high-level model which makes programming more efficient and more reliable by automating memory management and garbage collection.

- SML does static type checking which detects many errors at evaluation time. Error detection is enhanced by the use of pattern matching and by the exception mechanism.

- The SML module system is an extension of the underlying polymorphic type system thereby providing separation of interface specification and implementation. These facilities are very effective in structuring large programs and defining generic, reusable software components.

Chapter 4

Latency Insensitive Protocols

In this chapter, we present two different latency insensitive protocols: Relay-station based and Bridge-based protocols. We provide the functional definitions of the protocols, along with code snippets of their models in SPIN and SML.

4.1 Relay-station based protocol

The relay-station based protocol borrows the idea of relay stations and signal pipelining from the earlier proposed protocol by Carloni et al. [70, 69]. In their protocol, logic blocks called relay stations are placed along the wires where the signal propagation distance greater than one clock cycle. These relay stations work as pipeline blocks to send data from one shell to another over a long wire. They consist of two registers that work like a FIFO. Each process (called *perl*) is encapsulated in a wrapper to form a *shell*. The wrapper consists of three main processes: an equalizer, a stall signal generator and an extended relay station. Their equalizer process consists of queues placed on each input with a logic that synchronizes the inputs. The stall signal generator creates stall signals based on the inputs received by the equalizer. The extended relay stations (also called *half-relay stations* [73]) are placed on each output signals of the perl to enable queuing at the output of the wrapper.

We present a simplified version of the relay station based approach which includes the relay station along with the shell wrapper. Our wrapper mainly consists of the equalizer that includes the functionality of the stall signal generator, thus simplifying the protocol proposed in [68]. Furthermore, there is no need for extended relay stations at the outputs.

The refinement steps for transforming a synchronous system to a latency insensitive system can be thought of as a two stage operation. The first stage involves encapsulating the synchronous components and the next stage involves refining the interconnects to make them consistent with the flow of events (Figure 2.3.1 of Chapter 2). In the first stage, each module is encapsulated with an equalizer. An equalizer is a process instantiating template that given n input signals and a stall signal, it produces n output signals and n stall signals.

4.1.1 Formal Description

Let V be the set of data values and, T be a countable set of time stamps. Unless otherwise specified, in this section, we assume T = $\mathbb{N}$ = set of natural numbers. An event $e \in V \times T$ is an occurrence of a data value with a particular time stamp. However, in the systems we consider, a special event called *absent event* denoted by τ that may occur[1]. Therefore, the set of all events is denoted by E, where $\tau \in E$ and for all other $e \in E$, $e \in V \times T$. When $e \in V \times T$ it is called an *informative event*. A *signal* s is defined to be a sequence of events, often denoted as $e_1e_2e_3\dots$ where $e_i \in E$.

For the preliminary definitions, if s is a signal, $s[i]$ denotes the i^{th} event, hence either $s[i] \in V \times T$ or $s[i]$ = τ. The set of all signals is denoted by S. There are input signals, output signals and *stall* signals. A stall signal s_t is a sequence of boolean events, i.e., $s_t[i] \in Bool \times T$. The set of all stall signals is denoted by S_T. In our system, IPs are hardware modules that map input signals to output signals, therefore we refer to them as processes. A process p is a function $S^n \rightarrow S^m$ where $n, m \in \mathbb{N}$. A synchronous system consists of these processes where communication and computation happens at the global clock. The communication among these processes is assumed to be zero-delay and each process takes one cycle for computation.

In the remainder of this section, we define a few terms and notations that are used in the chapter.

Definition 1. *Given $s \in S$ and $e \in E$, we define $e \oplus s = s'$ where $s' = e :: s$, s.t. e is the first element and s is the rest of the signal.*

Definition 2. *Given one tuple of m elements and another of n elements, $\bigodot$ creates a tuple of $m + n$ elements. $< a_1, \dots, a_n > \bigodot < b_1, \dots, b_m >=< a_1, \dots, a_n, b_1, \dots, b_m >$*

Definition 3. *Given two tuples of n events and n signals respectively, $\bigoplus$ creates a tuple of n signals with an event appended to each signal.*

$< e_1, \dots, e_n > \bigoplus < s_1, \dots, s_n >$ = $< e_1 \oplus s_1, \dots, e_n \oplus s_n >$

Definition 4. ***Latency Equivalence**: The two signals s_1 and s_2 are said to be latency equivalent, $s_1 \equiv_e s_2 \Leftrightarrow \mathcal{F}(s_1) = \mathcal{F}(s_2)$, where*

[1] It may be caused due to lack of valid data in the producer or due to the consumer's request to delay a transmission

$\mathcal{F} : S \rightarrow S$ be defined as, $\mathcal{F}(s) = \sigma(s, 1, n)$ and,

$$\sigma(s,i,n) = \begin{cases} \sigma(s, i+1, n), & if\ s[i] = \tau \\ s[i], & if\ (i = n) \\ s[i] \oplus \sigma(s, i+1, n), & otherwise \end{cases}$$

$\mathcal{F}$ takes a signal s as input and outputs a signal s' that contains no τ events, but preserves all informative events. The helper function σ takes the signal s, n which is the length of the signal s and the initial index 1 as parameters. σ is defined recursively with the following cases: If the event at current index is τ, then σ is called with the index incremented. If the event is not τ and the index reaches the length of the signal, then σ terminates by returning the last event, otherwise the informative event at the i^{th} position is returned with σ called to check for the next event.

Definition 5. ***Sequential composition****: Given two processes p_1: $S^u \rightarrow S^v$, p_2: $S^v \rightarrow S^w$ and $s_1, \cdots, s_u \in S$, we define the sequential operator $\circ$ as:*

$$p_2 \circ p_1(s_1, \cdots, s_u) = p_2(p_1(s_1, \cdots, s_u))$$

Definition 6. ***Feedback composition*** *[135]: Given a process p: $(S \times S) \rightarrow (S \times S)$ and $s_i, s_j, s_k \in S$, we define the feedback operator $FB_p(p)$ as:*

$$FB_p(p)(s_i) = s_k \textit{ where } p(s_i, s_j) = (s_j, s_k)$$

The signal s_j is an internally generated signal and the behavior of the feedback process is defined using fixed point semantics [135]. For unique fixed point to exist, we assume all processes to be monotonic and continuous. For simplicity, we define the feedback composition for a specific process with two input and output signals, though it can be easily generalized for processes with multiple inputs and outputs.

Definition 7. *Vectorization function $\Upsilon_{i=1}^{n}(exp(i))$ evaluates the expression $exp(i)$ for i from 1 to n.*

$$\Upsilon_{i=1}^{n}(exp(i)) = < exp(1), exp(2), \cdots, exp(n) >$$

where, $exp(k)$ is a textual replacement of i by k in $exp(i)$.

An equalizer ($\mathcal{E}$) is a process that given n input signals and a stall signal, it produces n output signals and n stall signals. The functionality of the equalizer can be divided into three modes:

1. *Disable mode*: In this mode, the equalizer is stalled by another process through an input stall signal. The equalizer sends absent events on all its output signals and enables all the output stall signals using function *InsertStl* (shown in Definition 8).

2. *Absent mode*: In this mode, the equalizer receives an absent event on one of its input signals and its input stall is disabled. The equalizer sends absent events on all its output signals and stalls only those processes from which it received an informative event using function *InsertAbt* (shown in Definition 8).

3. *Present mode*: The equalizer receives informative events on all its input signals and its input stall is disabled. It places these informative events on the output signals using function *InsertEvt* (shown in Definition 8).

Definition 8. ***Equalizer** ($\mathcal{E}$) : Given $s_1, \ldots, s_n \in S$ and $s_t \in S_T$, the equalizer $\mathcal{E}$: $(S^n \times S_T) \rightarrow (S^n \times {S_T}^n)$ is defined as:*

$$\mathcal{E}(s_1, \ldots, s_n, s_t) = eval(s_1, \ldots, s_n, s_t, 1, \ldots, 1)$$

where,

```
$eval(s_1, \ldots, s_n, s_{t1} :: s_{t2}, i_1, i_2, \ldots, i_n)$ =
  if $(s_{t1} = false)$ then
    if $(\exists_{j=1}^{n}\ (s_j[i_j]) = \tau)$ then
      $InsertAbt \bigoplus evalnextindex$
    else $InsertEvt \bigoplus evalnextevent$
  else $InsertStl \bigoplus evalnextstall$
```

$InsertAbt = < \tau, \tau, \ldots, \tau > \bigodot \Upsilon_{j=1}^{n}(exp_1(j))$

$InsertEvt = \Upsilon_{j=1}^{n}(s_j[i_j]) \bigodot < false, \ldots, false >$

$InsertStl = < \tau, \tau, \ldots, \tau > \bigodot < true, \ldots, true >$

$evalnextindex = eval(s_1, \ldots, s_n, s_t, \Upsilon_{j=1}^{n}(exp_2(j)))$

$evalnextevent = eval(s_1, \ldots, s_n, s_t, \Upsilon_{j=1}^{n}(i_j + 1))$

$evalnextstall = eval(s_1, \ldots, s_n, s_t, \Upsilon_{j=1}^{n}(exp_2(j)))$

$exp_1(j)$: *if* $(s_j[i_j]) = \tau$ *then* $false$ *else* $true$

$exp_2(j)$: *if* $(s_j[i_j]) = \tau$ *then* $i_j + 1$ *else* i_j

The equalizer is defined using a helper function $eval$ that takes n signals $s_1, ..., s_n$, a stall signal s_t and initial indices for each input signal $1, ..., 1$ and returns n signals and n stall signals. The initial indices are given assuming that the first event for each signal is at that position.

4.1.2 An Example

Let us consider an example shown in Figure 4.1. In the example, there are two synchronous processes P and Q that are encapsulated with their corresponding equalizers EqP and EqQ. We call this encapsulation P' and Q'. There is a delay of two cycles on the interconnection between P' and Q'. Therefore, one relay station is placed on the interconnect to permit successful transfer of data from P' to Q'. The solid arrows denote the data signal whereas the light dotted lines denote the stall signals. The process P' reads data from the environment and process Q', performs its functionality, and provides the data on its output which connects the relay station. On the following clock cycle, the relay station outputs the data to Q'. After Q' computes, the data is output on its two output signals. One of its outputs connect back to P', and other provides data to the environment. All the components are connected by a single clock. If valid data is not present on any of the input signals of a process, then the process is stalled. We assume that each data signal is associated with an additional valid signal (not shown) which indicates whether the data is valid or invalid. For each process, its stall signals are enabled/disabled by its destination processes. This is because if a process is not ready to accept valid values, it can disable its previous processes to stop them from producing new values. This feedback mechanism is prevents the usage of infinite buffers. We assume that the processes considered here are deterministic functions, and are continuous as well as monotonic [138, 135].

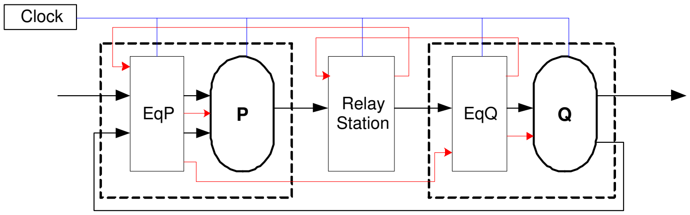


Figure 4.1: Relay-station based latency insensitive protocol

4.1.3 SPIN Model

In PROMELA, an equalizer process and its composition with its original process is modeled. The PROMELA code of an equalizer for a process with two inputs is shown in Listing 1. The SPIN model of the equalizer consists of two temporary buffers for each signal to store the values on the incoming signals. We also declare variable $valid_i$ which keeps the track of the number of informative events in the buffer for $signal_i$. In every clock cycle the equalizer reads a value during the read phase and store the value in the temporary variable. As the value is stored, the $valid_i$ is incremented for that signal. In the write phase of the same clock cycle, the value is written on the output provided the buffer count (i.e $valid_i$) is greater than 1 for all signals. Otherwise an absent event is placed on all output signals. The logic for the output stall signals of the process are based on the number of informative events on the buffer.

Listing 1: PROMELA code for Equalizer

{Comment: The code for the equalizer is presented below. $eventtype_A$==1 means that event on $signal_A$ is informative. There are two temporary buffers for each signal. $temp1_A$ is the first temporary buffer for signal A and $temp2_A$ is the second temporary buffer for signal A. The variable $stall$ == 1 denotes that the equalizer is stalled.}

```
proctype Equalizer() {
    int $valid_1$, $valid_2$;
loop:
    Synchronize reading with other processes.
                /* START OF READ PHASE */
    if
    □ ($eventtype_A$ && $valid_1$ == 0) → $valid_1$++;
          Store value in $temp1_A$
    □ ($eventtype_A$ && $valid_1$ == 1) → $valid_1$++;
          Store value in $temp2_A$
    □ (!$eventtype_A$ && $valid_1$ == 1) → $temp1_A$ = $temp2_A$
    □ else → skip
    fi;
    if
    □ ($eventtype_B$ && $valid_2$ == 0) → $valid_2$++;
          Store value in $temp1_B$
    □ ($eventtype_B$ && $valid_2$ == 1) → $valid_2$++;
          Store value in $temp2_B$
    □ (!$eventtype_B$ && $valid_2$ == 1) → $temp1_B$ = $temp2_B$
    □ else → skip
    fi;
    Synchronize writing with other processes.
                /* START OF WRITE PHASE */
    if
    □ ($valid_1 > 0$ && $valid_2 > 0$ && !$stall$) →
          $valid_1$–; $valid_2$–;
          Place events on output signals
    □ else → Place absent events on output signals ;
    fi;
    Set output stall signals based on $valid_i$ values
goto loop; }
```

4.1.4 SML Model

In this section, we discuss the SML model for the latency insensitive protocol for simulation-based validation. In SML, a finite signal is modeled as generic list, whereas an infinite signal is written as delayed function application as shown in Listing 4.1.

Listing 4.1: Finite and Infinite Signal

```
(* Definition of a finite signal *)
datatype signal = nil | 'a :: 'a list

(* Definition of an infinite signal*)
datatype infseq = nil | cons of 'a * (unit -> infseq)
```

For convenience, we formulate an event to be a list of two elements, where the first element is the value and the second element identifies whether the event is an informative event or an absent event (eg. e_j = [3,1] is the j^{th} event with 3 as the value and 1 as the identity of the event[2]). We can also model an event as a tuple of two elements, however, we prefer to use a list instead due to the ease of their built-in manipulation functions provided in SML. A signal can then be formulated as a list of events. (eg: s_i = [[1,1],[2,1],[3,0],. . .]).

Listing 4.2 shows the implementation of the $equalizer$ process in SML. The equalizer reads one event from all the input signals of a process along with an event from the stall input. It then checks if all the events at a time are informative. The check for events is done through the $etypes$ and $info$ functions (lines 6-13). The functionality setting the stall values for *Disable mode* is done by the $stallon$ function and the output is given by $e3$ (line 21). The stall values when the equalizer is in *absent event mode* is set by $stallset$ function and the output is given by $e2$ (line 20). Finally, the *valid mode* output is given by $e1$ (line 19).

Listing 4.2: Equalizer

```
1  fun equalizer() = fn s => fn st => f(s,st,indexstart(length(s))
2
3  fun f([],st1::st,_) = [] | f(_,[],_) = [] | f(_,_,[]) = [] |
4      f(s,st1::st,i) =
5  let
6   fun etype(x1::x2) = x2| etype([])=nil
7   fun etypes[] = [] | etypes(x1::x) = etype(x1) @ etypes(x)
8   fun info [] = false |
9   info(x1::[]) = if (x1 = 1) then true else false |
10  info(x1::x) = if (x1=1) then ( info(x)) else false
11  val allevents = e(s,i) (* Events from all the signals at a time *)
12  val allinfo = if info(etypes(allevents)) = true
13    then true else false (* True when all events are informative *)
14  fun stalloff(0) = [] | stalloff(n) = [1] @ stalloff(n-1)
15  fun stallon(0) = [] | stallon(n) = [0] @ stallon(n-1)
16  fun flipval(x) = if x=1 then 0 else 1
17  fun stallset([]) =[] | stallset(x1::x) = [flipval(x1)] @ stallset(x)
18
19  val e1 = [allevents,[stalloff(length(allevents))]]
20  val e2 = [tauevents(length(s)),[stallset(tags(allevents))]]
21  val e3 = [tauevents(length(s)), [stallon(length(allevents))]]
22 in
```

[2] 1 corresponds to an informative event and 0 corresponds to an absent event

```
(case(st1) of
 1 => (if allinfo = true
   then ([e1] @ f(s,st,incrementindex(i)))
   else ([e2] @ f(s,st,incrementempty(i,etypes(allevents))))
      ) |
 0 => ([e3] @ f(s,st,i)) |
 _ => [])
end
```

The $equalizer$ process is then sequentially composed with the synchronous process to form the shell of the process.

Listing 4.3: Shell Encapsulation

```
fun shell() = fn s => fn st => Increment() (#1(Punzip() (Pequalizer() s st)))

fun unzip() = fn s => (unzipsig(s),stallval(unzipstall(s)))
```

4.2 Bridge-based protocol

In this section, we propose a new latency insensitive protocol, which disposes of the relay-stations by adding some interface logic and wiring. While this approach requires more wires, it does not increase the number of components that have to be placed, and therefore uses less iterations for the placement and routing. The relay-station based approach requires multiple place and route iterations to estimate the number of relay-stations required to be placed on the long wire. Furthermore, these relay stations are to be placed at strategic points to ensure that the signal propagation of each wire segment is less than the clock period.

4.2.1 Protocol Description

In this protocol, instead of placing relay stations along the long interconnect that connect two modules, we place a *bridge* at the interface of the modules. The basic idea of our approach is that if a event between two components takes two clock cycles, we only send one value every second clock cycle thereby adapting the communication speed between components to their distance. An outgoing interface takes care of this restriction, and stalls the sending component whenever it delivers values too fast. While this eliminates all relay stations, it however slows down the system significantly due to an increasing number of stalls. We eliminate this slowdown by additional communication lines. The number of interconnects needed depend on the interconnect delay. For example, if there is an interconnect delay of n cycles, then n interconnects are placed in-between the processes for each wire. Note that in most cases n is in the range of 2 or 3.

These additional wires are an extra cost; however in modern processes it is less expensive to add parallel wires than to add components such as relay-stations which take additional area. However, one may argue that these long interconnects may cause loss of signal integrity. This may not be true in this case as our motivation is driven from the fact that IP clock speeds have increased and the interconnect distances have remained constant. We assume that if the clock speed of IPs is reduced, then the same interconnect would not be considered long. Furthermore, to allow these multiple interconnects to be packaged successfully, multi-metalayer hierarchy of interconnect levels can be used. Traditional packaging approach uses a two-level interconnection, however these can go up to 10 levels of Cu interconnects for 90nm-generation logic devices [182, 191].

The bridge based protocol requires insertion of a bridge process on the long interconnects, and a *bridge* process is a composition of two processes: a *splitter* at the output of a “shell” and a *merger* at the input of the other “shell”. The *bridge* not only ensures correct flow of events from one process to another, but also ensures that the delay in between the events is minimized. Each *bridge* process has one input signal and one output signal. The block diagram of the *bridge* process is shown in Figure 4.2. To ensure that the events correctly transfer from one process to another through long interconnects, we add extra interconnects that are bounded by a *splitter* on the source and a *merger* on the sink.

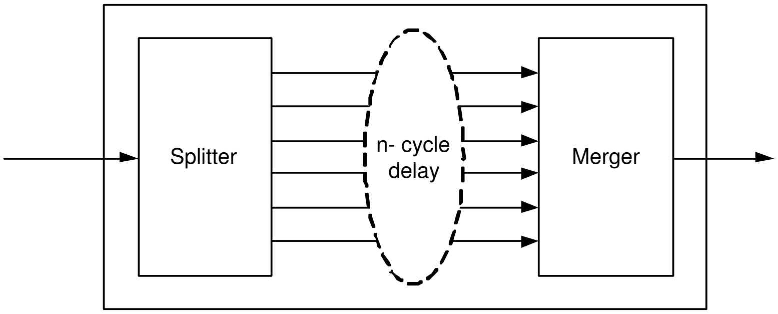


Figure 4.2: latency insensitive system with Bridge

The $splitter$ and the $merger$ process are connected by n interconnects where n is the delay on the long interconnect. Hence, $splitter$ process has n output signals. This process contains simple placement logic for the placement of events on these n signals. The splitter is implemented at the output of a process, and it transfers events on the corresponding signal. The splitter only places one event on one of the output interconnects and absent events are placed on the rest of the signals at a particular time stamp. Assuming that there are i events on the input signal of the splitter, at every cycle, the i^{th} event is placed on the n^{th} signal based on a rotational scheme. For example, if the delay on the interconnect is 3 cycles, then in the first cycle, the first element will be placed on the first signal and absent events will be placed on the other two signals. In the next cycle, the second event will be placed on the second signal and absent events will be placed on the first and third signals and for the third event it will follow the scheme. After the third event is placed, in the following cycle, the fourth event will be placed on the first cycle again. This rotation scheme will continue for the rest of the events. This functionality is illustrated by the definition shown below:

Definition 9. ***Splitter***
Given $s \in S$, the Splitter $\mathcal{H} : S \rightarrow S^n$ is defined as:

$\mathcal{H}(s) = spread(s, n, 1)$ *where,*

$$spread(x :: y, n, i) = \begin{cases} place(x, n, i, 1) \bigoplus spread(y, n, 1), & if\ i = n \\ place(x, n, i, 1) \bigoplus spread(y, n, i + 1), & otherwise \end{cases}$$

$$place(x, n, i, j) = \begin{cases} x \bigodot insertAbt(n - j), & if\ i = 1 \\ \tau \bigodot place(x, n, i - 1, j + 1), & otherwise \end{cases}$$

$$insertAbt(n) = \begin{cases} \tau, & if\ n = 1 \\ \tau \bigodot insertAbt(n - 1), & otherwise \end{cases}$$

The splitter is defined using a helper function ***spread(s,n,1)*** that takes three parameters which are the signal s, delay on the interconnect n and initial index of the signal s. *spread* uses the *place* function to send an event on the appropriate output signal. The function *place* puts τ on all signals using *insertAbt* except for the i^{th} signal on which it places the i^{th} event of the input signal.

Contrary to the splitter, we implement a *merger* that takes n input signals and outputs one signal. The merger also extracts one event from the input signals based on the rotational scheme as illustrated earlier and places it on the output signal. The functionality of the *merger* is formally defined below:

Definition 10. ***Merger***
Given $s_1, \cdots, s_n \in S$, the merger $\mathcal{M} : S^n \rightarrow S$ is defined as:

$\mathcal{M}(s_1, \ldots, s_n)$ = *ext($(s_1, \ldots, s_n)$, n, 1) where,*
$ext((x_1 :: y_1, \ldots, x_n :: y_n), n, i) =$

$$\begin{cases} rem((x_1, .., x_n), n, i) \oplus ext((y_1, .., y_n), n, 1), & i = n \\ rem((x_1, .., x_n), n, i) \oplus ext((y_1, .., y_n), n, i + 1), & otherwise \end{cases}$$

$$rem(x :: y, n, i) = \begin{cases} x, & if\ i = n \\ rem(y, n, i + 1), & otherwise \end{cases}$$

The merger is defined using the helper function ext that takes as parameters the signals $s_1, \ldots, s_n$, delay of the signal n and the index of the first signal. *ext* extracts the informative event from the appropriate signal and places it on the output signal using the *rem* function. *rem* returns the event at the i^{th} position.

4.2.2 SPIN based description

The model of a latency insensitive system with the bridge-based approach was modeled and verified in SPIN in our validation framework described in chapter 5. We formally verify the assertion for latency equivalence using the SPIN model checker for an example with two processes as shown in Figure 4.3. The example is of a simple parity checker with two processes, P and Q, where based on non-deterministic input, process P produces a value based on its input. Process Q checks if the

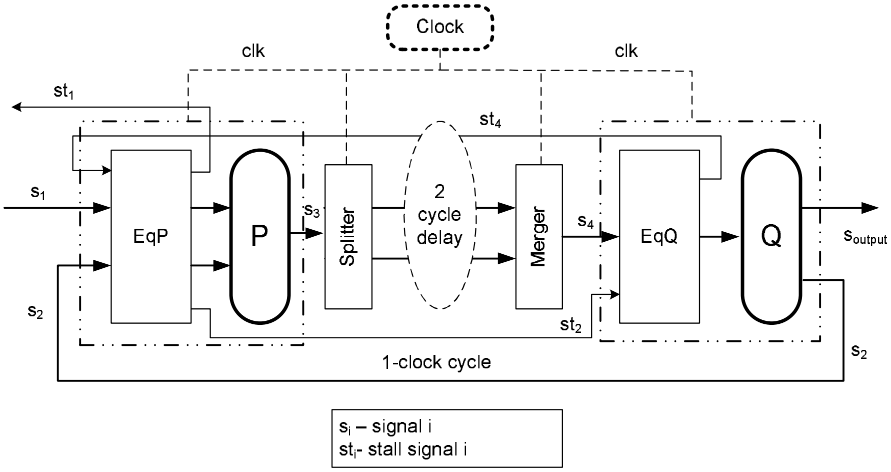


Figure 4.3: Latency Insensitive System with Bridge

input is a 1 or 0, and outputs a corresponding boolean value accordingly. We use our validation framework (proposed in chapter 5) to ensuring the correctness of the protocol.

Listing A: PROMELA code for Splitter

{Comment: The splitter is placed at the output of a synchronous module connecting to a long interconnect. For this implementation we assume that the interconnect delay is two cycles. The synchronization is done when the module gives an output. The placement variable called $place$ is defined s.t when $place$ == 0 then the event is placed on $signal_0$, otherwise it is placed on $signal_1$ }

```
proctype Splitter() {
    int place=0;
loop:
    Synchronize with process.
    if
    □ $place$ == 0 → $place$=1;
            Place event on $signal_0$
            Place absent event on $signal_1$
    □ $place$ == 1 → $place$=0;
            Place event on $signal_1$
            Place absent event on $signal_0$
    fi;
goto loop; }
```

The PROMELA code of the splitter process is shown in Listing A. We assume that there is a two clock cycle delay on the interconnect where the splitter is placed. A temporary variable $place$ is defined that places the events from the input signals to either $signal0$ or $signal1$, depending on the current value of the $place$. These two signals connect the splitter and the merger. The $place$ variable keeps changing every clock cycle.

The PROMELA code of the merger process is shown in Listing B. Based on the logic used by the splitter, similar logic is used to read the values from the two incoming signals. The $place$ variable is offset in this module based on the delay on the interconnect. The values read from the signals are then placed on the output.

Listing B: PROMELA code for Merger

{Comment: The code for the merger is presented below. The merger is placed at the input of synchronous module on the long interconnect. The synchronization is done when the module reads. The $extract$ variable is defined s.t when $extract$ == 0 then the event is taken from $signal_0$, otherwise it is taken from $signal_1$ }

```
proctype Merger() {
    int extract=0;
loop:
    Synchronize with process.
    if
    □ $extract$ == 0 → $extract$=1;
            Extract event from $signal_0$
    □ $extract$ == 1 → $extract$=0;
            Extract event from $signal_1$
    fi;
goto loop; }
```

4.2.3 SML description

In this section, we describe the implementation of our bridge-based approach in SML. The SML implementation of the $splitter$ process is shown in Listing 4.4. An input signal and the interconnect delay is given to the $splitter$ process. One event is read from the input signal and $insertevent$ function (line 6) places the event from the input signal to one of the interconnects and absent events are placed on rest of the interconnects. The events are placed in the rotational scheme as illustrated earlier.

Listing 4.4: Splitter

```
fun splitter(n) = fn s => f(s,1,n)

fun  f([],_,_) = [] |
 f(x1::x, i, n) =
let
 fun insertevent(_,j,0) = [] |
     insertevent(y1,j,n) = (if n = j
   then [y1] @ insertevent(y1,j,n-1)
   else [[0,0]] @ insertevent(y1,j,n-1))
in
 if (i = n)
   then [insertevent(x1, i, n)] @ f(x, 1, n)
   else [insertevent(x1, i, n)] @ f(x, i+1, n)
end
```

The SML representation of the merger is shown in Listing 4.5. The $extractevent$ function extracts one event from all signals at a time (line 3). Extraction of events from the signals is done in similar way as they are placed on the interconnects by the splitter.

Listing 4.5: Merger

```
1  fun merger(n) = fn s => g(s,n,1)
2
3  fun g([], n, i) = [] | g(x1::x, n, i) =
4  let
5   fun extractevent([],n) = [] | extractevent(x1::x,n) =
6   (case (n) of
7     1 => x1 |
8     _ => extractevent(x, n-1))
9  in
10  if (i = n)
11    then [extractevent(x1,i)] @ g(x, n, 1)
12    else [extractevent(x1,i)] @ g(x, n, i+1)
13 end
```

The splitter and merger are composed together. The input sequence of the splitter and the output sequence of the merger are equivalent, since the order of events written by the splitter on the n output signals and the order of events read by the merger from its n input signals is the same. Therefore, the flow of events from the output of one shell across the long interconnect to the input of the corresponding shell is maintained.

Now, bridge delays its input by n cycles. The delay on the bridge is modeled by the $Delayproc$ process that just delays the events by n cycles, where n is the delay on the interconnect (Listing 4.6).

Listing 4.6: $Bridge$ process

```
fun Bridge(n) = fn s => Delayproc(n) (merger(n) (splitter(n) (s)))
```

We now show, how we compose the processes P and Q to form a shell. We define intermediate processes $PunzipP$, and $PunzipQ$ to strip the values of the data, valid and stall signals from the equalizer. These striped values are then provided to the process. The SML code for this composition is shown in Listing 4.7.

Listing 4.7: Shell of process P and Q

```
fun Pshell() = fn s1 => fn s2 => fn st =>
 P() (#1(PunzipP() (Pequalizer() s1 s2 st)))
   (#2(PunzipP() (Pequalizer() s1 s2 st)))

fun Qshell() = fn s => fn st => Q() (#1(PunzipQ() (Qequalizer() s st)))

fun PunzipP() = fn s => (unzipsig1(s),unzipsig2(s),formatstall(unzipstall(s)))
fun PunzipQ() = fn s => (unzipsig(s),formatstall(unzipstall(s)))
```

We then compose the two shells and the Bridge to get $LIsystem$ process. This process outputs the result of the system when the two inputs are provided to the shell of P along with the stall signals for both the shells. Using this process, we can then computed the output of the system with $LIcompute$ process which takes in two inputs at shell of P and its stall signal. Listing 4.8 shows the implementation of the processes $LIsystem$, $LIcompute$ and $Pshellstall$. The function $Pshellstall$ gives the stall values for shell of Q.

Listing 4.8: Latency insensitive system

```
fun LIsystem() = fn s1 => fn s2 => fn st1 => fn st2 =>
   Qshell() (Bridge(3) (Pshell() s1 s2 st1)) st2;

fun LIcompute() = fn s1 => fn s2 => fn st1 =>
 (LIsystem() s1 s2 st1 (Pshellstall() s1 s2 st1),
   LIsystem() s1 s2 st1 (Pshellstall() s1 s2 st1),
     Pshellstall() s1 s2 st1);

fun Pshellstall() = fn s1 => fn s2 => fn st =>
 (#3(Punzip() (Pequalizer() s1 s2 st)));
```

The final step involves taking care of the feedback signals. In functional languages, the feedback is defined by the fixed-point semantics. For each evaluation cycle[3], the feedback signal has to be computed again. Listing 4.9 shows the fixed point computation for the feedback semantics.

Listing 4.9: Feedback Process

```
fun fb(p) = fixpt(p,s,[],length(s)+1)
(* The fixpoint is computed on event basis *)
fun fixpt(q,s,sout,0) = sout | fixpt(q,s,sout,n) =
 fixpt(q,s,(q s sout), n−1)
```

4.2.4 Case Study: Adaptive Modulator

We consider a case study of an adaptive modulator that consists of three IPs: regulator, convolutor and analyzer. The regulator module takes an input signal and a control signal and outputs based on the control signal by adding a threshold value. This output is then multiplied with a masking value by the convolutor module. The output of the system is given by the amplitude signal. The analyzer module outputs the control signal based on the input of the amplitude. The connections of these components are shown in Figure 4.4.

[3] In each evaluation cycle, a process consumes an input and produces an output.

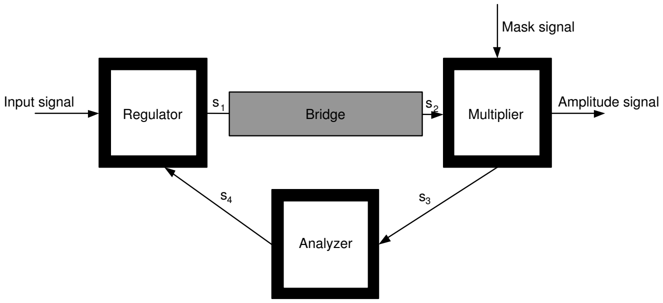


Figure 4.4: Adaptive Modulator

With these three IPs, we follow the refinement steps described in Chapter 2 to construct the latency insensitive system. We have these three synchronous components. Early floor planning and interconnect routing is done to find the delays on the interconnects. We assume in this case that the regulated input signal is a long interconnect. Hence, we encapsulate the modules with the equalizer and repeat floor planning to find the delay on the interconnects. The long interconnect is then refined by adding a bridge process. The new latency insensitive representation of the adaptive modulator is shown in Figure 4.4. The SML implementation is done using the components described in the previous section. In order to check the correctness of the latency insensitive model, we create a model containing the latency insensitive implementation as well as its synchronous counterpart with the synchrony assumption on its interconnects. We feed the same input sequence to both models and verify the latency equivalence of their outputs as described in Chapter 5.

The case study presented here is just an example to show how the latency insensitive refinement of a synchronous system can be done and validated against its original implementation. Any deterministic functionality can be integrated in a synchronous module and can be compared with its latency insensitive refinement. Since, changing the functionality is easy in a functional framework, many case studies can be analyzed successfully. On the other hand the functionality cannot be changed and verified easily when the system is being formally verified using a model checker.

In order to check the correctness of the latency insensitive system, we setup the two systems as described by our validation framework. We feed the same input sequence to both models and validate for the latency equivalence of their outputs. We have implemented this latency insensitive system for a finite signal input as well as for an infinite signal input. For finite signals, we can see the output of the $Eqcomparator$ process for as many input events given. In the case of infinite signals, we can check for the desired number of input values as computation for infinite values is based on delayed function application.

4.3 Multiclock Extension to Latency Insensitive Protocol

The latency insensitive systems proposed earlier have been mainly targeting single clock synchronous systems where all components operate on the same clock. We now consider extending the existing latency insensitive implementation for multiclock systems where different components with different clocks are connected via arbitrarily long interconnects. The need for a system with components having different clocks arises when different IP blocks from different vendor are integrated in the same system. At this time, however, we are only permitting the use of components with defined clock relations, also called rationally clocked systems. By clock relation, we mean that there is a known ratio of the evaluation cycle[4] between different components. In the SML framework, the notion of clock is defined by the evaluation cycle of the processes. This approach therefore makes it possible to connect rationally clocked systems.

We modify our original refinement methodology for multiclock refinement. Before encapsulation of the processes, we add an $Insert$ and a $Strip$ process to each synchronous component of the system. The $Insert$ process inserts n absent events for each event on the original incoming signal where n is the ratio of events on the incoming signal to the number of events evaluated by the process in each cycle. The output of the $Insert$ process is then given to the original process. The formal definition of the $Insert$ process is shown below:

Definition 11. *$Insert$ is a process, s.t. $\mathcal{I}(s) = s'$ where*

$$s' = g(y, n) \textit{ and,}$$

$$f(n) = \begin{cases} \tau, & if\ n = 1 \\ \tau \bigodot f(n-1), & otherwise \end{cases}$$

$$g(x1 :: x, n) = (x1 \bigodot f(n)) \bigoplus g(x, n)$$

We also place a $Strip$ process at the output of the synchronous component. This $Strip$ process removes the extra absent events inserted by the $Insert$ process. The formal definition of the $Strip$ process is given below:

Definition 12. *$Strip$ is a process, s.t. $\mathcal{W}(s) = s'$ where*

$$s' = g(y, n) \textit{ and, } t(x1 :: x) = x$$

$$f(s, n) = \begin{cases} t(s), & if\ n = 1 \\ f(t(s), n-1), & otherwise \end{cases}$$

$$g(x1 :: x, n) = f(x1, n) \bigoplus g(x, n)$$

[4] In each evaluation cycle, a process consumes an input and produces an output.

Once these processes are composed with the original component, we can then follow the refinement methodology. In SML, we can easily modify our aforementioned latency insensitive system to an latency insensitive system containing components with different evaluation cycles. The SML implementation of the two processes is show below:

Listing 4.10: Multiclock Interface

```
1  fun Insert(n)= fn s1 => h(s1,n)
2  fun h([],_) = [] | h(x1::x, n) =
3  let
4   val sig1 = [x1] @ tausall(n)
5  in
6   [sig1] @ h(x,n)
7  end
8
9  fun Strip(n) = fn s1 => f(s1,n)
10 fun f([],_) = [] |
11  f(x1::x,n) =
12 let
13  fun dr [] = [] | dr(x::xf) = xf
14  fun drop ([],_) = [] | drop(s,1) = dr(s) |
15    drop (s,i) = drop(dr(s),i−1)
16 in
17  drop(x1,n) @ f(x,n)
18 end
```

The $Insert$ process appends absent events to an event received from the $equalizer$ process (lines 2-7). The number of absent events depend upon the ratio of the evaluation speed of the process and the rate at which the inputs are received from the $equalizer$ process. Similarly, the $Strip$ process receives one output event with extra absent events appended every evaluation cycle of the system (lines 10-18). The extra absent events are discarded by the $Strip$ process. Both these processes are composed with the shells. The process $Insert$ is applied to all the input signals of the process. The parameter n represents the number of cycles used by the process compared to a single communication clock of the system. The SML implementation is shown in listing 4.10.

Chapter 5

Validation Framework

Given various latency insensitive protocols (LIPs), it is essential to ensure the correctness of the protocols that are being composed for latency insensitive designs. There are various techniques that can be used to check the correctness of the design. Simulation based validation techniques can be used to check if the desired output is achieved for given set of inputs. Although dynamic validation is appropriate for flushing out protocol design errors, such validation only covers certain input sequences. Furthermore, a latency insensitive design is capable of receiving inputs which may be delayed by an arbitrary time, which can not be validated using dynamic techniques. Another way to confirm the correctness of such an implementation is to mathematically formalize it, as done in [70]. But mathematically proving the equivalence of two systems is a challenging task and not beyond mistakes. It requires complex mathematical proofs that are not straightforward to follow by others who want to confirm them. Furthermore, every new variation of latency insensitive protocols cannot be validated easily using mathematical proof techniques. Model checking techniques can also be used to ensure correctness of a latency insensitive design; however, in the case of an error, it may be difficult to identify whether the error was cause due to an incorrect design or an erroneous protocol. Furthermore, due to the subtleties involved in the optimizations, it is plausible that the newly invented LIPs have serious flaws. We have experienced this in our attempts to optimize Carloni's protocol [68, 69]. As a result, we felt that there is a need for a framework where these protocols can be quickly modeled and validated.

In this chapter, we propose such a validation framework for families of latency insensitive protocols, both for dynamic validation, useful for early debug cycles, and formal verification for formal proof of correctness. For formal verification, we use the SPIN model checker [130] to verify the correctness of an LI system, whereas for the simulation based technique, we use SML [20] for quick validation. We compare and contrast the two techniques and find that the SML based simulation for validation is a more convenient way to validate the protocols, especially for debugging the early versions of the protocols. This can be a useful framework in the hands of designers trying to create new latency insensitive protocols or to optimize existing ones for design convergence.

In order to formally verify such protocols, the LI system as well as the synchronous idealization have to be modeled formally, and the latency equivalence has to be captured as a formal property. The best way is to provide designers with an easy to use framework to model and validate their protocols. In this thesis, we provide a validation framework for families of latency insensitive protocols. We present validation techniques and frameworks for ensuring correctness of latency insensitive designs [206, 207, 214].

5.1 Framework Description

In the framework, we model the latency insensitive system along with its synchronous idealization and provide the same input signals to both the systems. These input signals can also be latency equivalent. The block diagram of the framework is shown in Figure 6.4. In the figure, the environment provides the designs with two inputs and the designs produce two outputs. The framework can be modeled for any number of inputs and outputs. The framework along with a synchronous design and its latency insensitive counterpart consists of a sender environment, a receiver with a comparator, and a feeder module.

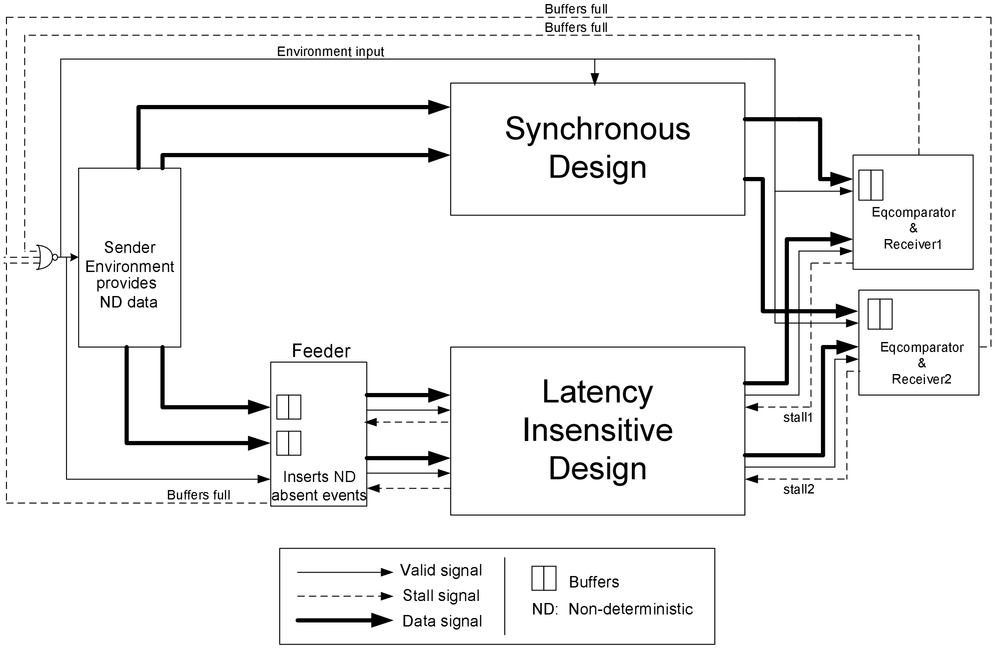


Figure 5.1: Validation Framework

- **Sender Environment:** The sender environment provides non-deterministic data to both the designs. Note that same inputs are provided to both designs. The sender is modeled as a synchronous module that can be stalled based on its input control signal. A high input control signal indicates that the sender module can execute, otherwise it is stalled. This control input is a NOR of the buffer full signal from the feeder and all the buffer full signals from the eqcomparators (based on the number of outputs). For the figure shown, the control input signal is a NOR of three signals: two from the eqcomparators at the outputs and one from the feeder. Each of these signals indicate whether the storage buffers in the corresponding modules are full or not. An enable signal (high value) indicates that the buffers are full, and hence a NOR for any high signal would result a low output disabling the sender environment.

- **Synchronous Design:** The synchronous design is assumed to consume and produce valid data on every cycle. The synchronous design along with its inputs from the environment has an environment signal. This signal indicates that the data is valid at its inputs and can be gated with its clock. When the clock arrives, and the environment signal is valid, the synchronous design executes and produces a valid output.

- **Feeder:** The feeder module takes inputs from the sender environment and provides them to the latency insensitive design. This module is mainly used during formal verification to insert non-deterministic absent events on the signals to the latency insensitive design. This insertion of non-deterministic absent events ensures checking the correctness for unbounded delay for the latency insensitive design. However, for formal verification, fairness constraints are imposed on the delays. This means that that valid data will eventually be placed on the signals to the latency insensitive design, and the delay will not be infinite. This is due to the basic assumption that the data is not lost due to the delay. The control signal of the sender environment is also provided to the feeder, indicating valid data is produced by the sender environment. The feeder makes a non-deterministic choice of either storing this data into its buffers and inserting absent events on its outputs, or forwarding the data to its outputs. The valid data is stored in the buffers of the feeder module, which act like a FIFO with two storage elements. The two storage buffers are sufficient for ensuring that the data is not overridden. The stall signal is enabled when one of the buffers becomes full. By the time the stall signal is read by the sender environment, the next data produced which is written in the second buffer location. The feeder outputs data and valid signal to the latency insensitive design. A high on the valid signal indicates valid event, otherwise it is an absent event. The feeder also takes stall signals from the latency insensitive design. These stall signals indicate that the latency insensitive design is not ready to accept data on its corresponding data input signals, and the feeder outputs absent events on these.

- **Latency insensitive design:** Application of latency insensitive protocol to the synchronous design yields a latency insensitive design. This latency insensitive design is the design being verified against its synchronous counterpart. For every input and output data signal, there is an associated valid and stall signal.

- **Eqcomparator:** The $Eqcomparator$ module is a reduced version of the an $equalizer$ module (discussed in chapter 4). Similar to the equalizer, the $Eqcomparator$ process reads the informative events from the output signals of the two designs (synchronous and latency insensitive). The informative events on these signals are compared and checked to be latency equivalent. In the case when an absent event is seen on one of the output signals, it is discarded and the next event is considered on the same signal. The informative events on the two output signals are compared in sequence to ensure correct functionality. This is done by putting an assertion in the $eqcomparator$ process. Appropriate buffers are placed to store the values from the synchronous design. When the buffers are full, a stall signal is send to the sender to stop it from producing new values. Listing 5.1 shows the SML code for the comparator.

Listing 5.1: Eqcomparator

```
fun Eqcomparator() = fn s1 => fn s2 => compare(s1,s2,1,1);

fun compare([],_,_,_)= [] |
 compare(_,[],_,_)= [] |
 compare(s1,s2,i,j) =
let
val event1 = extractevent(s1,i);
val event2 = extractevent(s2,j);
val valid = if event1 = [] orelse event2 = [] then false else true;
val valpresent =
 if valid = true
   then if tag(event1) = [1] andalso tag(event2) = [1]
     then true else false
 else false;
fun comp(x,y) = if (x=y) then true else false;
in
 if valid = true andalso valpresent = true
    then (if comp(value(event1),value(event2)) = true
     then [true] @ compare(s1,s2,i+1,j+1)
     else [false])
   else if valid = true
     then if (tag(event1) <> [0] andalso tag(event2) <> [0])
     then [false]
   else if (tag(event1) = [0] andalso tag(event2) = [0])
     then compare(s1,s2,i+1,j+1)
   else if (tag(event1) = [0] andalso tag(event2) = [1])
     then compare(s1,s2,i+1,j)
   else if (tag(event1) = [1] andalso tag(event2) = [0])
     then compare(s1,s2,i,j+1)
   else [true]
 else [true]
end
```

- **Receiver:** The receiver module receives the data from the two designs. In the framework, the receiver module is combined with the $Eqcomparator$ module. The receiver module also provides non-deterministic stall signals to the latency insensitive design.

In contrasting the two techniques, we find formal verification to be useful when we want to exhaustively check the system for correctness for all possible paths. This approach may be time consuming but would ensure complete validation of the system. On the other hand, the SML based simulation had its own set of advantages. We found SML based simulation validation to be an easier way to find the bugs in the protocol at an earlier phase of the design process by simulating the framework with a set of test vectors and checking for the correctness of the system. Also, due to the inherent denotational semantics of functional languages, we found it easier to formalize such a framework. We realized that the formal definitions of the components of LIP could be naturally mapped to SML. Hence, it was easy to model the framework in SML. Also, the component of the LIP were made generic such that they could easily be reusable with any component. It also helped in making the models open to extension without making many changes.

Chapter 6

Formal Verification of Elastic Architectures

In this chapter, we show how we establish the correctness of synchronous elastic flow (SELF) protocol [89, 88]. Application of the SELF protocol to a synchronous design yields an elastic design. An elastic design is another term for latency insensitive design. The SELF protocol is briefly discussed in Chapter 3. The protocol works by replacing all the flip-flops of a design by elastic buffers, and appropriate fork and join structures are used depending on the number of input and output signals.

We want to ensure that the design resulting from the application of the SELF protocol is elastic. The definitions for an elastic architecture are presented in [148]. We take the implementation details of the control of the elastic buffer, join and fork structures from [89], and verify that they satisfy the definitions of the elastic machines. Furthermore, to ensure data consistency, we verify that the transfer streams of the elastic machines are latency equivalent to the transfer streams of its synchronous counterpart. This verification is done in a *Transfer Trace (TT)* validation framework for elastic machines (similar to our validation framework). We initially use the SPIN model checker [130]. SPIN targets efficient software verification and is primarily used for asynchronous systems, however, we use it for verification of synchronous systems. Its friendly nature of manipulating arrays allows for parameterization of the components based on number of inputs and outputs. The properties are specified as LTL and assertions within code. We also model and verify the elastic controllers using the NuSMV model checker [82] which is more amiable to hardware verification. However, the NuSMV model checker is unfriendly towards handling and manipulating arrays. Since, we want to parameterize the design, we pre-process the model based on the number of inputs and outputs of the design using the GNU m4 preprocessor [6].

This work primarily highlights the following tasks:

1. Verification of an elastic module composed by a m-input join, an elastic buffer, and a n-output fork structure, against the properties of elastic machine. (SPIN & SMV)

2. Verification for latency equivalence of an elastic design with its synchronous counterpart.

(SPIN & SMV)

3. Verification of variable latency unit for the properties of elastic machine. (SPIN)

6.1 Elastic Module

In this section, we discuss the details of the elastic module. An elastic module results from replacing the flip-flops of a design with elastic buffers, and adding appropriate fork and join structures that provide support for its multiple inputs and outputs. Figure 6.1 shows the application of the SELF protocol to a non-elastic design to create an elastic module. In the figure, the flip-flop is replaced by an elastic buffer, and join and fork structures are added to the input and output interface of the elastic buffer. The source modules are considered as the producers and the destination

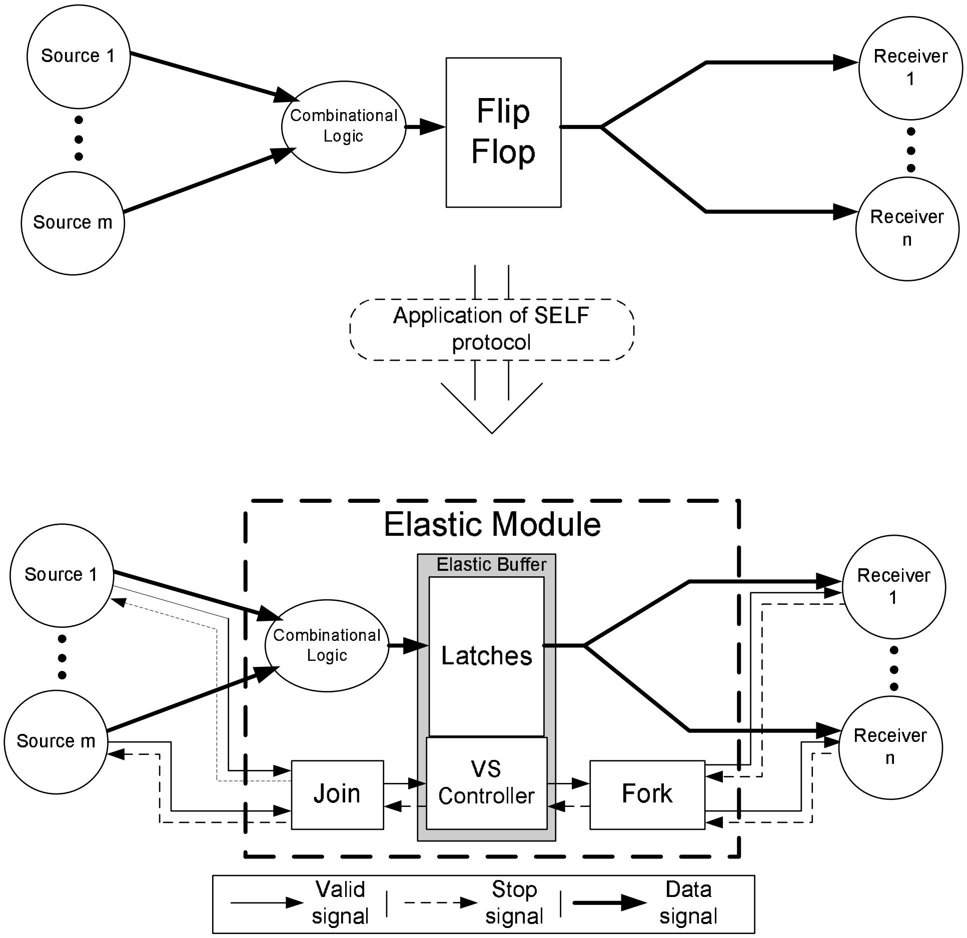


Figure 6.1: Application of SELF protocol

modules as consumers. A valid transfer between the two modules occurs when valid signal is high and stop signal is low. The data is retransmitted in the case when both the stop and valid signals are high. The elastic module asserts the stop signal to its producers only when either its storage elements are full or valid values are not realized on other signals. The combinational logic represents the functionality on the data inputs. This input is only accepted and stored in the elastic buffer when high value is seen on all input valid signals.

6.1.1 Elastic Controllers

In an elastic architecture, each data signal is associated with a valid and a stop signal. These three signals form a channel. We say that a transfer occurs on a channel if the valid signal is high and stop signal is low. A high valid signal indicates the presence of valid data on the data signal. Each elastic buffer has a single input channel and a single output channel. An elastic buffer is divided into two parts: data and control. The control part controls the flow of data through the data part. The data signal connects to the data part, where as the valid and stop signals connect to the control part. Many different control policies can be associated with the elastic buffers. For example, in W_1R_1 control policy, the data is always written to the first latch and the data is read from the second latch, whereas in W_2R_1 data can be written either in the first latch or second latch, and read from the second latch [89]. The promela code for the elastic buffer with W_1R_1 control policy is given in Listing 6.1. The code is divided into two phases: high phase and low phase. The high phase represents the positive edge of the clock, whereas low phase represents the negative edge. In the data part, there are two latches: $highlatch$ and $lowlatch$ to store data, which are controlled by the control signals $ctl\_highlatch$ and $ctl\_lowlatch$ respectively. In the control part, there are four latches: $H1$ and $H2$ which are updated in the high phase, and $L1$ and $L2$ which are updated in the low phase. The input data, valid and stall signals are represented by $datain$, $validin$ and $stallin$, and the corresponding output signals are represented as $dataout$, $validout$ and $stallout$. The behavior of the elastic buffer is given below.

Listing 6.1: Promela code for Elastic Buffer

```
/*********  High Phase  *********/
/*  Control part */
H1 = L1 & L2;
H2 = L1 | L2;
ctl_highlatch = validin & !H1;
ctl_lowlatch = L1 & !L2;

/* Data part */
if
:: ctl_lowlatch -> highlatch = lowlatch;
:: else -> skip;
fi;

/*********  Low Phase  *********/
/* Control part */
L1 = validin | H1;
L2 = stallin & H2;
ctl_highlatch = validin & !H1;
ctl_lowlatch = L1 & !L2;
```

```

/* Data part */
if
:: ctl_highlatch -> lowlatch = datain;
:: else -> skip;
fi;

/**** Output ****/
validout = H2;
stallout = H1;
dataout = highlatch;
```

Next, we look at the join and fork structures. The join structure is used for inputs and a fork structure is used for outputs. There can be various implementations for joins and forks such as lazy/eager joins, and lazy/eager forks. The usage of appropriate joins and forks should be such that there exist no combinational cycles when these structures are composed together.

Since the join structure is placed at the input of the control part of the elastic buffer, its inputs are valid signals from its source modules, and a stop signal from its elastic buffer. The outputs of the join are the stop signals to its source modules, and a valid signal to its elastic buffer. The promela code for the parameterized lazy join structure is given in Listing 6.1. Assuming there are M input signals, each i^{th} input valid signal is denoted as $validin[i]$, and the corresponding output stop signal is denoted as $stopout[i]$. At its interface with the elastic buffer, it provides a $EBvalidin$ signal as the input valid signal to the elastic buffer, and receives a stop signal from the elastic buffer denoted as $EBstopout$. The behavior of the lazy join is given below:

Listing 6.2: Promela code for Lazy Join

```
i=0;
do
:: i==0 -> EBvalidin = validin[i];
   i++;
:: i<M && i>0 -> EBvalidin = EBvalidin & validin[i];
   i++;
:: else -> break;
od;

if
:: M==1 -> stopout[0]=EBstopout;
:: else ->
   i=0;
   do
   :: i < M ->
      stopout[i] = (validin[i] & !(!EBstopout & EBvalidin));
      i++;
   :: else -> break;
   od;
fi;
```

The fork structure is placed at the output interface of the elastic buffer. Here, we consider an eager fork which has a latch associated for each output. Assuming we have N destination modules, the input to the fork are N input stop signals denoted as $stopin[i]$ for $0 \leq i < N$, and their corresponding output valid signals denoted by $validout[i]$. The latches are denoted as $Fork[i]$ for the corresponding outputs. $EBvalidout$ and $EBstopin$ are the valid and stop signals at the interface of the elastic buffer.

Listing 6.3: Promela code for Eager Fork

```
if
:: N==1 -> EBstopin = stopin[0];
:: else ->
   i=0;
   EBstopin = stopin[i] & Fork[i];
   i++;
   do
   :: i < N ->
      EBstopin = EBstopin | (stopin[i] & Fork[i]);
    i++;
   :: else -> break;
   od;
fi;


if
:: N==1 -> validout[0] = EBvalidout;
:: else ->
   i=0;
   do
   :: i < N -> validout[i] = EBvalidout & Fork[i];
      i++;
   :: else -> break;
   od;
fi;

/* Updating */
if
:: N==1 -> skip;
:: else -> i=0;
   do
   :: i< N ->
      Fork[i] = (stopin[i] & Fork[i]) | !(EBstopin & EBvalidout);
      i++;
   :: else -> break;
   od;
fi;
```

The verification model built was designed to be reconfigurable for the number of inputs and outputs to the elastic machine. This modeling style realized a push button model for verifying the design with m inputs and n outputs.

6.1.2 Properties for Elastic Machines

For an elastic machine, three main properties are defined for its channels: *persistence*, *forward liveness* and *backward liveness*. The persistence property is defined for all the channels of the elastic machine. This property states that if there is a valid value being transmitted and a stop is seen on the channel at the same time, then the same valid value will be retransmitted in the following cycle. This property ensures that the data is not lost when a stop is seen. It should re-transmit until the stop is disabled. A disabled stop and a valid value on a channel means that a valid transmission has occurred.

The liveness properties ensure that the hungriest channels[1] in the elastic machine are not left starving. These properties are defined with respect to other input and output channels of the elastic machine. The forward liveness property ensures that the hungriest channels at the output of the elastic machine are provided with valid data eventually. The backward liveness property ensures that the hungriest channels at the input of the elastic machine will eventually have its stop disabled.

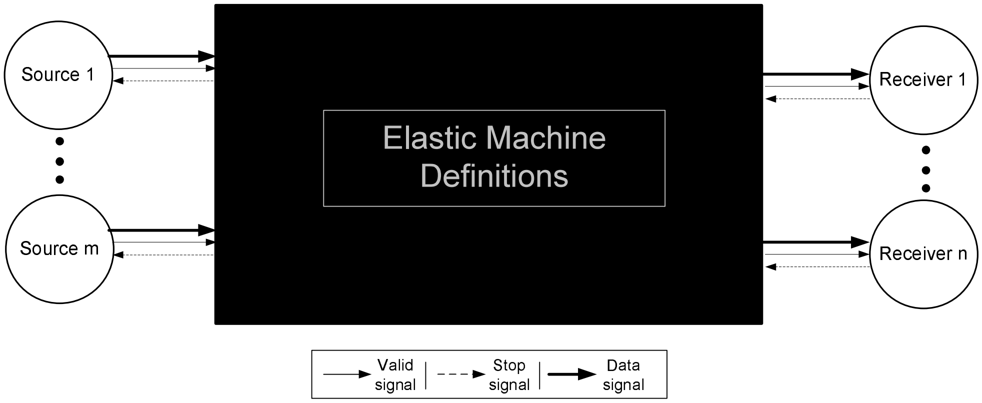


Figure 6.2: Elastic Machine Blackbox

As liveness properties ensure that the hungriest channels of the machine are not starved, a comparison of the number of transfers on the channels is done to identify the hungriest channels. A token count variable tct is used to keep track of the number of token transfers. For the properties shown below, Y is considered to be an output channel in the output channel set O, X is considered to be an input channel in the input channel set I. The properties for an elastic machine are given below [148].

1. Persistence: [] $(valid_Y \wedge stop_Y \rightarrow X\ valid_Y)$.
2. Forward Liveness: [] $(min\_tct_O \geq tct_Y \wedge min\_tct_I > tct_Y \rightarrow <> valid_Y)$
3. Backward Liveness: [] $(min\_tct_{I \cup O} \geq tct_X \rightarrow <> \neg stop_X)$

6.1.3 Verification

The elastic module is modeled and verified using both SPIN and NuSMV model checkers. The producers and the consumers are modeled as environments. The environments provide the elastic machine with non-deterministic values for valid and stop signals. Each data value provided to the elastic module is a sequence of numbers generated from 0 to a constant Ct. The combinational

[1] A channel with the least number of transfers as compared to other channels

logic is of a simple adder. For this specific verification task, our input is a sequence of numbers, and the adder is used as the combinational function. This helps in writing assertions to ensure that data is correctly processed. The environment is also made elastic such that it reacts appropriately to the elastic module. The elastic environment re-transmits the same data in the following cycle when the elastic machine is stopped. Property verification and assertions were used to verify for correct outputs.

During verification, it was realized that for backward liveness to satisfy, the fairness constraints are needed to ensure that the valid values are fairly seen on the other input channels. This is because, the join structure stops the input channel when the valid values were not seen on other channels. In SPIN, the properties were expressed as LTL, whereas, in NuSMV, they were expressed as CTL. Since, the liveness properties require taking the count of the number of transfers on each channel to identify the hungriest channel, this count can go up to very large integer possibly infinite. In the next section, we describe how we model this token counter in a finite domain.

Token Counter

The liveness properties for an elastic machine are expressed in terms of token counts on the channels. These properties assume infinite transfers on its input and output channels. However, given the current state of the tools used for formal verification, model checking such properties for such a design is almost infeasible. Therefore, we transform this model to a finite domain model by restricting the counter size of tokens. We introduce two counter variables which we call α count and β count to compare the token counts. Both these counter variables are dependent on the *synchronous distance* of the module. The synchronous distance denotes the maximum capacity of the module.

Definition 13. *The capacity $C(i, j)$ is defined as the maximum number of data storage elements between the two channels i and j.*

For example, given a simple elastic buffer (as defined earlier) with an input channel s_i and output channel s_j, its capacity C(s_i, s_j) = 2 (Recall that elastic buffer had two latches for data storage). The capacity of a channel is 0. The capacity can only be defined on a path where the token passes from one channel to another. With many channels, a module can have different capacities for different channel combinations. Given different capacities between channels of a module, we define its synchronous distance.

Definition 14. *For a module ϕ, and its input channels I and output channels O, its synchronous distance $\forall x \in I$ and $\forall y \in O$ is defined as $max(C(x, y))$. We denote the synchronous distance as $Syn(\phi)$.*

The synchronous distance denotes the maximum latency path from an input channel to an output channel. We set our count size to be equal to $Syn(\phi)$ of the elastic module.

Theorem 1. *For a module ϕ, and its token counts tct_i and tct_j for some input channel $i \in I$ and some output channel $j \in O$, $\mid tct_i - tct_j \mid \leq Syn(\phi)$.*

Proof sketch: For an elastic machine, the token count tct_i is incremented whenever a token transfer happens at the input channel i. This new transfer can only occur if there are empty storage elements in the module in the path of i. For some output channel j, an upper bound can be realized if all the storage elements, $C(i, j)$, are empty, and there is no output token transfer on channel j. This upper bound is then equal to the capacity of $C(i, j)$. So, we get $(tct_i - tct_j) \leq C(i, j)$. However, this capacity $C(i, j) \leq max(C(x, y)$, and $C(i, j) \leq Syn(\phi)$ implies $(tct_i - tct_j) \leq Syn(\phi)$. Similarly, the token count tct_j is incremented whenever a transfer takes place at the output channel. When the token is placed on the output channel, the storage elements connecting this output channel become free. Furthermore, a token can only be placed on the output if there are tokens present in the storage elements connecting to this output. Assuming all the storage elements have a token, the maximum number of tokens than can be placed on the output is the capacity $C(i, j)$ (assuming channel i has no new incoming inputs). We get $(tct_j - tct_i) \leq C(i, j)$. Since, $C(i, j) \leq max(C(x, y)$, and $C(i, j) \leq Syn(\phi)$ implies $(tct_i - tct_j) \leq Syn(\phi)$. Therefore, combining the two cases, we get $\mid tct_i - tct_j \mid \leq Syn(\phi)$. □

Theorem 1 shows that the synchronous distance gives the maximum difference of token transfers between any two channels of the module. We use this synchronous distance as our count size for counting token transfers on the input and output channels. As said earlier, we define two count variables, α count and β count in our model, the α count increments for every token transfer up to modulo synchronous distance. For a channel s_k, its α count for a token transfer is $\alpha_{s_k} = (\alpha_{s_k} + 1)\ mod\ Syn(\phi)$. This means that α count increases up to $Syn(\phi)$, and re-initializes. Now, we have successfully reduced the infinite count to a finite count of size $Syn(\phi)$, however, this does not completely solve our problem. We now need to compare the different counts on different channels. So, we now use β count variable to keep track of the relative counts with respect to all channels. This count updates for all the variables after every cycle, based on the transfer of tokens on any channel, and the value of the corresponding α count. Therefore, we only have these two counter variables in our model.

To show how these counters work, let us consider the scenarios of the elastic module shown in Figure 6.3. The elastic module has two inputs ($in1, in2$), and two outputs ($out1, out2$) with a synchronous distance of 5. So now, the α count for each channel counts up to modulo 5. The size of the β count is also equal to 5. In $senario\ i$, the actual token count on channels $in1, in2, out1$, and $out2$ are 3, 4, 3, and 2, respectively. Now taking modulo 5, gives us the α count of these channels (same as actual counts). However, β count updates with a window size of 5, and count 0 is allocated to the channel with least count. This channel with the least β count of 0 can also be referred to as the *hungriest* channel. The other channels are updated based on their difference of token counts with this hungriest channel. Therefore, the β count of the channels in1, in2, out1, and out2 are 1, 2, 1, 0 respectively. Here, $out2$ is the hungriest channel. Consider, in the next scenario $i + 1$, the actual token count for all the channels increases. Now, as the α count is updated based on modulo 5, the α count for these channels become 4,0,4, and 3, respectively. Note that

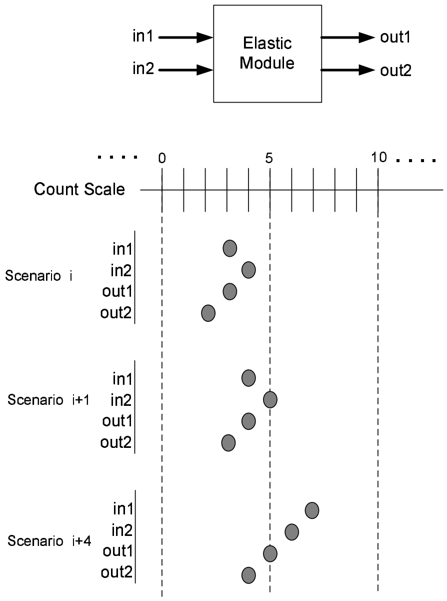


Figure 6.3: Scenarios for token counting

Channel	scenario i			scenario i+1			scenario i+4		
	actual	α	β	actual	α	β	actual	α	β
in1	3	3	1	4	4	1	7	2	3
in2	4	4	2	5	0	2	6	1	2
out1	3	3	1	4	4	1	5	0	1
out2	2	2	0	3	3	0	4	4	0

Table 6.1: Table showing how α and β count compare to actual count.

as the α count of channel in2 becomes 0, it becomes difficult to compare the relative counts of the channels. Based on the information of the α counts, we cannot identify their relative counts. However, the β count update tells us that out2 is the channel with least number of transfers. This helps us identify the the hungriest channel. Table 6.1 shows the actual, α, and β counts of the three scenarios of Figure 6.3. Another scenario after 4 cycles is shown as well, where channel in1 has the most number of token transfers.

Verification in SPIN

The elastic machine model is written in PROMELA, with three main processes: environment, combinational logic block and elastic buffer composed with join and fork structure. The friendly nature of using arrays and loops in PROMELA makes the parameterization of code simpler. The model is parameterized based on the number of inputs (I), number of outputs (O), the synchronous distance of the elastic module (Syn) and the bound of inputs provided to the combinational block ($[0, B]$). The sender environment generates the sequence of numbers modulo B as data input with the valid values. The receiver environment generates the stop values non-deterministically. New data value is only generated by the environment when a valid value is selected. Each signal is modeled as a shared variable. Conditional blocking statements are used in PROMELA for synchronizing the processes.

The combinational logic block of an adder is implemented. This module is also parameterized based on the number of inputs. The output of this module is read by the elastic buffer process. The elastic buffer with parameterized join and fork structures are written in one *proctype* based on how they are connected together.

Two token counts are defined for each channel: α count and β count. The α counter for each channel is incremented when a transfer for the channel occurred. Every time the counter reaches Syn, it is reset to 0 and the count restarts from 0. The β count updates at every cycle based on the least token count on the channels of the elastic module.

A single execution of the all the modules represents a single clock cycle for a synchronous design. Assertions are placed within PROMELA for ensuring that the sum computed, i.e the output of the elastic module, is correct. The persistence property is written as an assertion, where additional variables are used to store the value of the previous state. The liveness properties are written as LTL properties in the LTL property manager.

Verification in NuSMV

In NuSMV, each component behavior of the elastic module is written in a separate modules. Instances of these modules are composed together to get the desired behavior.

We use the m4 pre-processor to generate the NuSMV model based on the number of inputs and outputs. The user has an option of entering the four parameters for input, output, the synchronous distance, and bound of the data sequence generated by the environment. The m4 pre-processor creates the sender, receiver environments as well as the join and fork structure based on the number of inputs and outputs. The modules of the sender, receiver, join, elastic buffer, fork and the counter are written and composed together. The composed behavior is equivalent to its PROMELA model.

We specify the properties as CTL. These properties are also automatically generated based on the number of inputs and outputs. Both the model checkers yielded identical results (eg. fairness constraint requirement for satisfying backward liveness property).

6.2 Validation for Data Consistency

Data consistency for the elastic machines is checked in the validation framework presented in this section. The transfer streams of the elastic machine and its synchronous counterpart are checked for latency equivalence. Two streams are said to be latency equivalent if the order of their valid values is identical. The model of the framework is shown in Figure 6.4. The framework consists of three modules apart from the synchronous and elastic design blocks. The environment block provides same valid inputs to both synchronous and elastic designs whenever it is not stopped. As the environment is synchronous and non-elastic, an elastic feeder module is inserted before the elastic machine to non-deterministically inserts absent events to the input sequence provided to the elastic design. When absent events are provided to the design, the valid values are stored in the fifos of the feeder. Once the fifos are full, the feeder sends a stop signal to the environment to stop it from producing any more new values. The stop signals provided to the environment are also provided to the synchronous block. The synchronous block only produces valid values when it receives valid values from the environment. The fifos in the feeder consist of two storage elements with a W_2R_1 control policy [89].

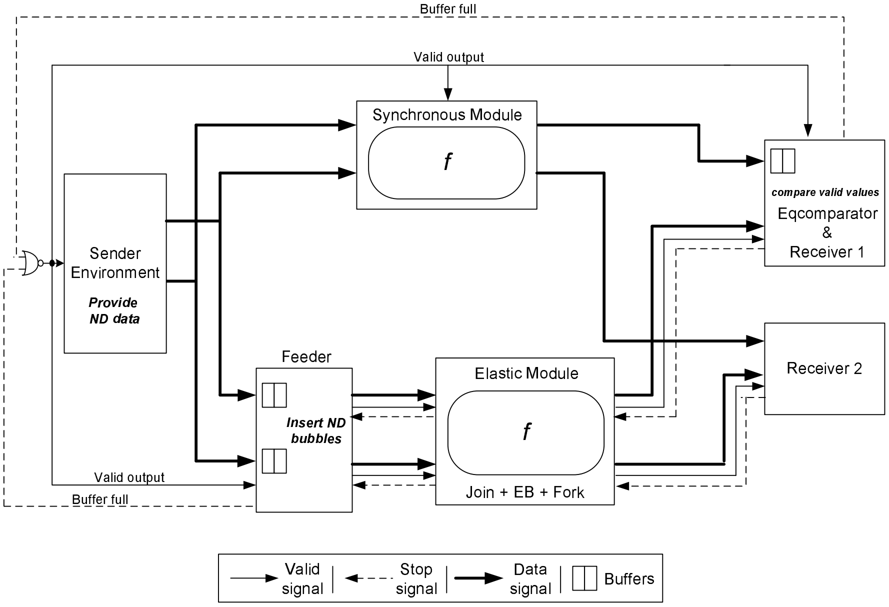


Figure 6.4: Transfer Trace Validation Framework

The module at the receiver of the elastic machine and the synchronous block is the $Eqcomparator$ module which compares the output of the two sequence to be latency equivalent. The $Eqcomparator$ module stores the values of the synchronous in a fifo if the output for the elastic machine is delayed. As the fifo becomes full, the environment and the synchronous blocks stops and no new value is produced. The stop is disabled when a value from the fifo is removed, which happens only when a valid data from the elastic machine is seen. The assumption is that the synchronous design produces the output in less number of cycles than the elastic design.

6.2.1 Combinational Logic as Uninterpreted functions

In the validation framework, we attempt to verify the design independent of the type of functionality placed in the combinational logic block. We use the notion of uninterpreted functions as proposed by Burch and Dill [65]. In their approach, the functions are modeled as uninterpreted function symbols, and the data is represented by terms. For their microprocessor model, the verification task involves checking the equivalence of the two nested terms. The same idea is also used by Bezerin et. al [48], where the microprocessor is modeled not to compute concrete values. The symbolic terms, which are made up of constants, and uninterpreted function symbols, are manipulated by the microprocessor. For symbolic model checking, a direct encoding of all possible terms and uninterpreted symbols is done, however it does not scale up with the number of instructions. This brute force method is not feasible to encode all possible terms when the number of instructions increases. Therefore, they employ a *reference file* with a compact encoding scheme based on only the terms that are possible in a single execution trace.

In our case, we employ the same procedure for modeling an uninterpreted function by encoding all possible terms. We use a single uninterpreted function for our case because the combinational logic providing inputs to the elastic buffer can be modeled as a single function. Furthermore, as we are comparing the outputs of the same functions in both the synchronous as well as the elastic modules, using the same uninterpreted function symbol is sufficient. Consider two inputs x_s, y_s for synchronous module, and other two inputs x_e, y_e for elastic module. For any deterministic function f, we know that

$$f(x_s, y_s) = f(x_e, y_e) \Leftrightarrow (x_s = x_e \wedge y_s = y_e)$$

In our case, as sender environment provides the same values to both the modules, we always have $(x_s = x_e \wedge y_s = y_e)$. Also, the elastic module provides valid output only when both its inputs are valid, therefore we should always get $f(x_s, y_s) = f(x_e, y_e)$. In our framework, we model check this property to be always true for valid values of both modules.

For uninterpreted function, we consider a two input uninterpreted function that takes two constants on each of its inputs. With two constants on each input, and one uninterpreted function, we get 8 possible terms for our model. We encode these 8 terms as shown in Table 6.2.

Intuitively, the same scheme can be applied for writing an uninterpreted function for any number of constants (data). As mentioned previously, the number of terms depend on the number of constants and uninterpreted functions. However, as our primary task here is to ensure consistent data transfer through an elastic module, it is sufficient to model check for our previous mentioned uninterpreted function because as long as the data provided to the two components is identical, the output of

X	Y	f(X,Y)
x0	y0	f(x0,y0) = z0
x0	y1	f(x0,y1) = z1
x1	y0	f(x1,y0) = z2
x1	y1	f(x1,y1) = z3

Table 6.2: Encoding for Uninterpreted function

the uninterpreted function will also be identical. Modeling checking for more number of terms becomes unnecessary.

6.2.2 Modeling in SPIN

Each component of the validation framework is modeled as a separate process. The environment process executes only if it is not stopped. Enumerated type is used for data values: x0, x1, y0 and y1. The data is selected non-deterministically for each signal where on one signal the possible values are x0 and x1, and on the other signal, the possible values are y0 and y1. The output values are also of enumerated type where z0, z1, z2 and z3 denote (x0,y0), (x0,y1), (x1,y0), and (x1,y1) respectively. The same combinational logic block was used for the elastic machine.

The feeder is modeled with two fifos, with a W_2R_1 control scheme, and a control for non- deterministically choosing to place invalid events on the output while storing the values. The implementation of the fifos is simple as the data is stored in the latch based on how many valid values are present in the fifo. The feeder is elastic and produces non-deterministic valid values when only it is not stopped by the elastic machine and has a valid value in its fifo buffers.

Assertion is placed in the $EqComparator$ module that checks whenever a valid transfer is seen on the elastic machine output. The assertion is the comparison of the two values.

6.2.3 Modeling in NuSMV

The NuSMV model is written similar to the PROMELA model. The components for the elastic buffer along with the join and fork structures are re-used from the earlier generated two input two output model from the pre-processor. The feeder module consists of two instances of fifo blocks with a control scheme of W_2R_1, and insertion of non-deterministic bubbles[2]. The block diagram of W_2R_1 structure is shown in Figure 6.5 .

The bubbles are inserted non-deterministically only when the fifo is not stopped and a valid value is present in the fifos. The finite state machine (fsm) and the definition of output signals are shown

[2] bubbles are invalid events denoted by valid signal being low

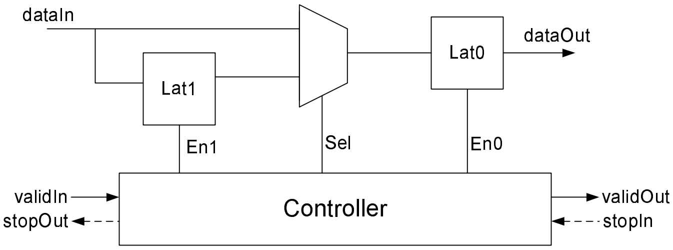
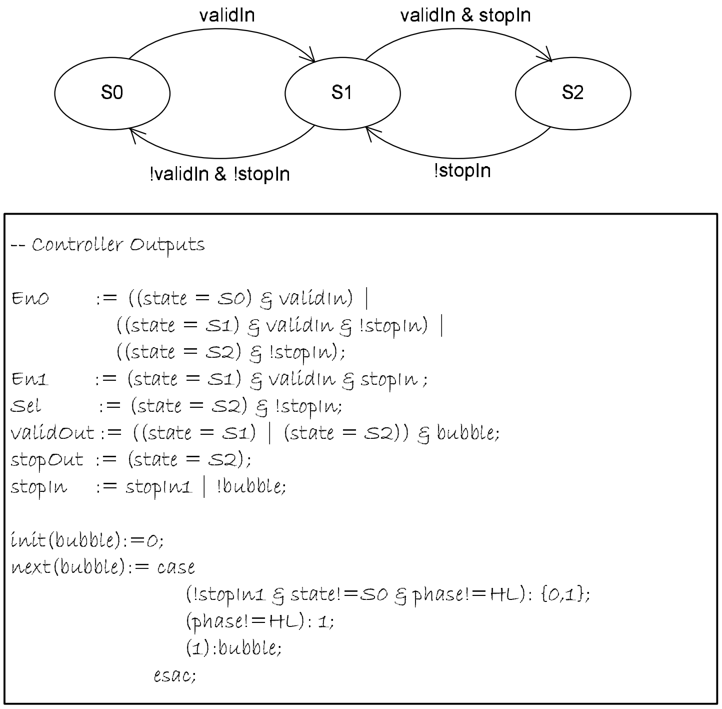


Figure 6.5: Block diagram of W2R1 fifo.

in Figure 6.6. There are three states in the fsm where S0 means empty, S1 means half full, and S2 stands for both full. The feeder took a single clock cycle.



```
-- Controller Outputs

En0      := ((state = S0) & validIn) |
              ((state = S1) & validIn & !stopIn) |
              ((state = S2) & !stopIn);
En1      := (state = S1) & validIn & stopIn ;
Sel       := (state = S2) & !stopIn;
validOut := ((state = S1) | (state = S2)) & bubble;
stopOut  := (state = S2);
stopIn    := stopIn1 | !bubble;

init(bubble):=0;
next(bubble):= case
                     (!stopIn1 & state!=S0 & phase!=HL): {0,1};
                     (phase!=HL): 1;
                     (1):bubble;
                  esac;
```

Figure 6.6: FSM and definitions of W2R1 control with ND bubble insertion.

The $EqComparator$ module also consists of a fifo, however, this fifo is optimized. It takes zero clock cycles in the case when it is not stopped. The block diagram of the fifo with two latches and two muxes is shown in Figure 8.7. These muxes select when to provide data at the output.

The fsm of the fifo is shown in Figure 6.8. The fsm has four states: S0, S1, S2 and S3, where S0 means that the fifo is empty, S1 means that data is in the second latch (Lat1) and the first latch (Lat0) is empty, S2 means that the data is in the first latch (Lat0) and the second latch (Lat1) is empty, and finally, S3 means that data is present in both the latches. Based on the data in the

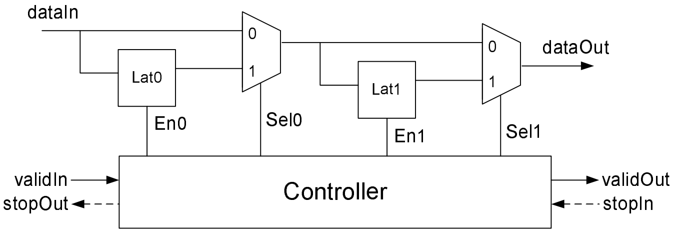
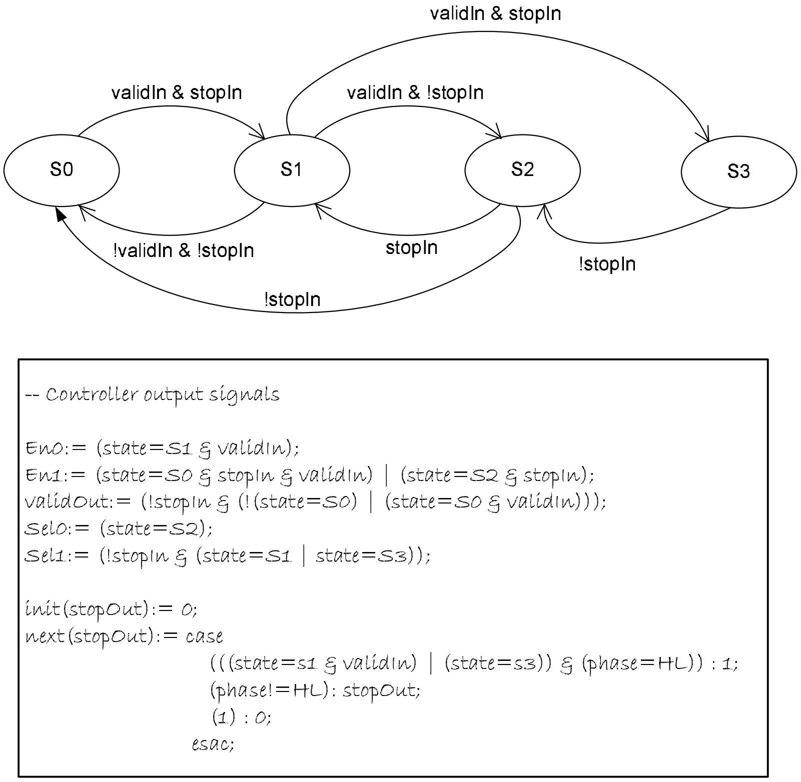


Figure 6.7: Block diagram of zero latency fifo.

latches, the stop signal is set for the next cycle.



```
-- Controller output signals

En0:= (state=S1 & validIn);
En1:= (state=S0 & stopIn & validIn) | (state=S2 & stopIn);
validOut:= (!stopIn & (!(state=S0) | (state=S0 & validIn)));
Sel0:= (state=S2);
Sel1:= (!stopIn & (state=S1 | state=S3));

init(stopOut):= 0;
next(stopOut):= case
                  (((state=s1 & validIn) | (state=s3)) & (phase=HL)) : 1;
                  (phase!=HL): stopOut;
                  (1) : 0;
                esac;
```

Figure 6.8: FSM and definitions of zero latency fifo.

The assertion is placed in the $EqComparator$ module that checks if the valid data on signal from the synchronous as well as elastic machine is same.

Inputs x Outputs	SPIN			NuSMV	
	States	Time(s)	Verification Mode	States	Time(s)
1x1	8504	0.2	Exhaustive	657	0.07
2x2	29232	0.36	Exhaustive	5193	0.22
3x3	673756	5.77	Exhaustive	41337	0.8
4x4	1.5E+07	196.35	Compression	329433	2.53
5x5	2.9E+08	3374	Bitstate :2	2.6E+06	12.38
6x6	5.5E+08	7794	Bitstate :2	2.1E+07	1544
7x7	6.3E+08	11622	Bitstate :2	1.7E+08	11005
8x8	6.5E+08	23474	Bitstate :3	1.3E+09	117967

Table 6.3: SPIN vs NuSMV

6.3 Variable Latency Model

The model of a variable latency unit is also verified using the SPIN model checker for the previously mentioned properties of the elastic machine. The backward liveness property failed to verify for the variable latency block without the fairness constraints on the input stop signal. The latency block is modeled to select a latency value non-deterministically. This non-deterministic value is decremented each cycle until it becomes 0, and then the valid value is placed on its output.

6.4 SPIN vs SMV

Although, the SPIN and NuSMV models are written based on the definitions of the elastic controllers, they both differ in the modeling approach. For example, in SPIN, the execution of each statement forms a new system state, whereas in NuSMV, a new system state is seen after the execution of all possible statements in a single cycle. Therefore, in SPIN each process was modeled as an infinite loop with synchronization variables placed at the end of the processes. A single loop of a process represents an execution of a single cycle. As a result of this, the persistence property is modeled as an assertion, where additional variables are introduced for denoting previous states.

A variable in SPIN can be assigned a value multiple times in a single cycle, whereas in NuSMV, the single assignment rule is enforced. Although, this made the modeling of the counters and fifos simpler in SPIN as compared to NuSMV, the model was more error prone with respect to a hardware model.

The number of reachable states for verification were more in SPIN as compared to NuSMV due to the fact that each statement execution was a new state. Table 6.3 shows the comparison of the reachable states and their respective verification times with respect to the number of inputs and outputs of the fork and join structures for SPIN and SMV. SPIN model checker ran out of memory after the verification of 4-input 4-output (4x4) elastic machine. For rest of the configurations bitstate verification mode was used with 2-bits used for representing a state from 5x5 to 7x7. For 8x8, 3-bits were used for bitstate verification which meant a lower coverage for verification.

Chapter 7

Trace-based Framework for IP Composition

In this chapter, we present our trace based framework that helps in analyzing types of systems that can be transformed to correct-by-construction globally asynchronous locally synchronous (GALS) designs.

7.1 Motivation

GALS as discussed in Chapter 1 is one of the solutions proposed to deal with the clock related issues that are seen in synchronous designs. A GALS design has the advantage of incorporating synchronous designs with asynchronous communication, where locally clocked synchronous islands communicate through point-to-point interconnections that have their own local timings. Furthermore, they allow us to isolate the clocks of different components making the clock skew problem irrelevant [93]. The future trend can be seen as designing systems with a GALS approach in mind. It is important to ensure that the IPs in GALS designs are provided with interfaces such that asynchrony in the communication is tolerated. However, there is a lack of good theoretical models and methodologies for designing GALS systems due to the fact that they involve both synchrony as well as asynchrony together. Furthermore, the validation of GALS designs are even more difficult.

Here, we advocate the use of existing synchronous tools for design and validation, however, we identify the types of systems that can be transformed to correct-by-construction GALS using a generic protocol. A system can be classified into one of the following clock frequency classes [74]: (i) Exactly matched clock frequencies (EMCF), (ii) Rationally related clock frequencies (RRCF), (iii) Nearly matched clock frequencies (NMCF) and (iv) Arbitrary clock frequencies (ACF). There is also an orthogonal classification of systems presented in [115, 168] as synchronous, plesiochronous, mesosynchronous, multi-synchronous, etc; however, in this chapter, we use the classification of [74]. A system classified as EMCF has all its components run on the same clock. The IPs are designed to assume that informative values are present on every signal at the arrival of the clock.

A value is *informative* if it is of relevance to a system. In real hardware, every block reads and writes informative values on their corresponding input and output signals at every cycle. If these values are to be ignored or older values are to be reused, techniques such as clock gating can be used [37, 62]. Next, we have systems classified as RRCF, where the components run on different but related clocks. In such systems, the clocks of all the components are related at the root of the *clock tree*. Let $C = \{c_1, c_2, \ldots\ c_k\}$ be the set of all the clocks for a system, and $w(c_i)$ be the frequency of the clock. An edge e is defined as $e = \{(c_i, c_j) : w(c_i)|w(c_j)\}^1$, where | is an acyclic relation and $c_i \neq c_j$. If $G = (C, E)$ is a directed graph, where E is the set of all edges, then we call it a *clock tree*. The frequency of the clock at the root of such a tree is obviously the multiple of every clock's frequency in the system. A *clock tree* of a system expresses the relationship of all its clocks. The sub-clocks are created generally by using frequency dividers/clock dividers. The clocks have sampling points at which the new events occur. We refer to these sampling points as *clock ticks*. Consider a producer and a consumer, where the clock frequency of the producer is twice that of the consumer. Now, the producer can produce two values in two clock ticks before the consumer can read them in its one clock tick. Next, we have systems with NMCF, also referred to as plesiochronous, where the frequencies are close but different. For example, in the design of network routers where each line card receives a bit stream. These streams are generated from different clock sources which are closely related in terms of their frequencies [74, 145]. The systems classified as ACF consist of clocks that are completely independent, and arbitrary. Such systems may have its components driven by some external clock sources. There are other classifications of systems such as mesosynchronous and multi-synchronous, where systems have same clock frequencies, however differ in the phases [115, 168]. In this work, we focus on the composition aspects for EMCF and RRCF systems, which are clocked by single clock or have rationally related clocks.

In the EMCF designs, all the IPs have the same clock frequency. In other words, on the arrival of the clock signal, the IPs execute; they read valid data from their inputs, and produce valid data on their corresponding outputs. So at every clock, valid data is produced and consumed. Now, when these are transformed to GALS, the IPs have independent clocks. However, we want to ensure that this GALS behavior is latency equivalent to its synchronous behavior. So one solution is to associate each inter-component signal with additional signals such that handshaking is done between the components to exchange data. Hence, the delay in the communication between the components would not change the behavior. The handshaking between the components would ensure that each IP would execute only when on all its inputs contain valid data, which was the case for the originally intended synchronous design.

Now consider the RRCF designs where IPs have related, but different clock frequencies. The same handshaking protocol will not work in this case as the clock frequencies are different. Each IP will have to sample data differently to ensure that the GALS behavior is latency equivalent to its synchronous counterpart. Hence, a special handshaking protocol would be required to ensure that the data exchanges are dependent on their related frequencies of their synchronous counterpart. So, for every different relation of clock frequencies, a new handshaking protocol will be required,

[1] | denotes divides.

which can be expensive in terms of their designing and ensuring their correctness.

We provide designers with a framework that helps in identifying the designs can be transformed to a correct GALS design which are mainly EMCF and RRCF. In our trace based framework, the behavior of a component is captured by its input and output traces. Also, the semantics of the composition are given by the composition of the traces. The synchronous as well as asynchronous traces can be seen in the same framework. We discuss these in the following sections. We define the property of *single activation* for a system, which means that for all IPs of the system, either all their the inputs and outputs have informative events at every clock tick or no informative events occur on any of them. In our framework, if the property of "single-activation" holds for a design, then it can be transformed to a behavioral equivalent GALS. Two behavioral-equivalent designs produce the same functional outputs for same inputs. We show behavioral-equivalence, by showing theoretically that systems with single activation are latency equivalent to their GALS design. By *latency equivalent*[2], we mean that the sequence of events carrying valid data on the output signals is identical [70]. Latency equivalence is an appropriate notion for behavioral equivalence, as we are only interested in the sequence of informative values in the system. As long as the order of valid informative outputs on the corresponding signals of the two systems are identical, they both are behaviorally equivalent. The single-activation property helps in identifying designs that can be transformed to GALS with barrier synchronizing inputs and avoiding complex handshake protocols. Now, if *single activation* holds for a design, then it can be transformed to a GALS design where its components can execute on independent clocks. This GALS design is correct with respect to its behaviors, and hence does not need to be verified again because we assume that EMCF and RRCF designs considered have been verified to be correct.

For our "correct-by-construction" transformation, we also highlight a barrier synchronization protocol for GALS. We use the example of an FIR filter to show how trace based framework can be used for validation and designing of a GALS system.

Please note that the main focus of this chapter is to show the formal basis of analyzing designs and their transformation to GALS. Hence, we do not compare with existing ad hoc GALS protocols. Furthermore, the presented protocol illustrates the idea of deploying such designs to GALS, and evaluation in terms of overhead, area and power may vary for different protocol styles.

7.1.1 Main Contributions of this Chapter

1. We provide a trace-based framework for identifying a class of IPs that can be composed to "correct-by-construction" GALS designs.

2. We show the transformation of RRCF designs to behaviorally equivalent synchronous designs.

3. We present the idea of a protocol that would enable this GALS transformation.

[2] We formally define latency equivalence in the following section

7.2 Preliminary Definitions

Let $\mathcal{V}$ denote the universal set of all variables. Each variable $v \in \mathcal{V}$ can be assigned a domain specific value, on an occurrence of an event. Let D be the set of all data values and, T be a partially ordered set (set of tags). We define three types of events: (1) informative clocked event e, where $e \in (D \times T)$, (2) informative un-clocked event $\bar{e}$, where $\bar{e} \in (D \times \mathbb{N})$, and (3) absent event τ, where $\tau \in (D_{\perp} \times T)$, and, $D_{\perp} = D \cup \{\perp\}$, where $\perp$ is a special element meaning that data is not present. The informative clocked events and the absent events are events that occur in a synchronous domain. For example, consider a synchronous adder, that adds two numbers when it is triggered. At every trigger, two informative clocked events are provided at the two inputs of the adder, and also an informative clocked event is produced at the output of the adder that contains the sum of the inputs. An absent event is seen when an informative clocked event is missed due to a delay on the interconnect at the trigger of the adder. Such an event does not have any relevant data value associated to it. In a real world design, such events can be notified by adding an extra valid signal associated with the data signal. Such valid signals are also used with latency insensitive protocols. On the other hand, the example of an informative unclocked event would be a value transferred by an asynchronous handshake between two components.

For each variable of the system, a series of data values are assigned which forms a trace. The trace of a variable shows the order in which values are assigned to a variable. This order of values assigned depends on events caused due to an input or result of some functionality. A trace can be finite or infinite.

Definition 15. *An* un-clocked trace $\overline{\sigma}(v) = \bar{e}_1, \bar{e}_2, \ldots$ *is a sequence of informative un-clocked events whose occurrence is associated with a variable* v*.*

Unclocked trace is seen for a variable that updates based some form of a handshaking scheme.

We now analyze a trace in a clocked domain, where events occur based on a clock[3]. In the clocked domain, the sequence number of the events correspond to a clock interval, which represents an observation point of an event. If we consider an observer who can look at all the individual clocks and their ticks, and align them using a global time scale, then each individual clocks can be analyzed w.r.t that time scale. The clock of the variable is inferred by its trace. The clock of a variable v is denoted by $\hat{v}$. In each clock interval of a variable, an event may occur. Let $\mathbf{C}$ be the set of all the clocks that can be associated to variables. For any variable v, its clock $\hat{v}$ contains the set of time instants at which its events occur.

We define a clocked trace as follows:

Definition 16. *A* clocked trace *(*σ*) is a sequence of either informative or absent events whose occurrence is associated with a variable* v*. Consider* $\sigma(v) = e_1\ \tau\ e_2\ \tau \ldots$ *where events are either informative or absent.*

[3] Given a timeline, a clock is the division of timeline into slots using boundaries

Example 1. *The* clocked traces *of variables* v_1 *and* v_2 *are represented as:*

$$\begin{array}{rccccccc}
 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 \\
\sigma(v_1): & e_{11} & e_{12} & e_{13} & e_{14} & e_{15} & e_{16} & e_{17} \\
\sigma(v_2): & e_{21} & \tau_2 & e_{23} & \tau_4 & e_{25} & \tau_6 & e_{27}
\end{array}$$

Here, $\hat{v}_1 = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$ *is a faster clock than* $\hat{v}_2 = \{t_1, t_3, t_5, t_7\}$.

For a trace σ of variable v, $\sigma(v)[i]$ denotes the value of the i^{th} event of the trace of v. For the above example, $\sigma(v_1)[5] = val(e_{15})$, where val is a function that returns the value of the event. An *empty trace* is a trace of a variable which is not associated with any informative event, and is represented by ϵ. An append operator $(@)$ is used to illustrate a trace creation. For example: $e_1\ e_2\ @\ e_3 = e_1\ e_2\ e_3$. The same applies to unclocked traces as well.

7.3 Analyzing Synchronous Designs

A *process* is a collection of traces, and can be classified as *synchronous* or *asynchronous*. A *synchronous process* has all its events indexed by a single clock. New events are computed for all its variables at every clock. A synchronous process can be recorded by a *synchronous* observation defined as follows.

Definition 17. Synchronous Observation*: An observation of a synchronous process,* P*, is defined as a 3-tuple* $\langle V, \Sigma, \hat{c}\rangle$*, where* V *is the variable set of* P*,* Σ *contains traces of all variables in* V *and* $\hat{c}$ *is the clock associated all the variables in* V*, i.e* $\forall v \in V$*,* $\hat{v} = \hat{c}$*.*

Example 2. *Consider an observation of a synchronous increment process* $\langle V, \Sigma, \hat{c}\rangle$*, where* $V = \{v_1, v_2\}$ *and* $\Sigma = \{\sigma(v_1), \sigma(v_2)\}$*. The increment process is defined as follows:* $v_1 = x$ *and* $v_2 = f(x) = x + 1$*. If the trace associated with* $\sigma(v_1) = e_{11}\ \tau_2\ e_{13}\ \tau_4\ e_{15}$ *with events* $e_{11} = (5, t_1),\ e_{13} = (7, t_3),\ e_{15} = (8, t_5)$ *then* $\sigma(v_2) = e_{21}\ \tau_2\ e_{23}\ \tau_4\ e_{25}$ *with* $e_{21} = (6, t_1),\ e_{23} = (8, t_3),\ e_{25} = (9, t_5)$*.*

Next, we look at *asynchronous* observations. An asynchronous observation can either be of a synchronous process or an asynchronous component such as a process that models memories, buffers, or asynchronous FIFOs with handshakes. An *asynchronous process* is not associated with any clock. Such a process reacts whenever an input is received. In the asynchronous observation, the information of clock is absent, and only informative events are recorded in the trace in their order of occurrences. Therefore, in an asynchronous observation, unclocked traces are considered for each variable.

Definition 18. Asynchronous Observation*: An asynchronous observation for a process written as* $(\overline{P})$ *is a two-tuple* $\langle V, \overline{\Sigma}\rangle$*, where* V *is the set of all variables of* $\overline{P}$ *and,* $\overline{\Sigma}$ *contains all* unclocked traces *(*$\overline{\sigma}$*) of variables in* V*.*

We define two forms of composition: asynchronous composition and synchronous composition. We first look at asynchronous composition of asynchronous observations.

Definition 19. *The asynchronous compositions of asynchronous observations ($\|^a$) for processes, $\overline{P}_1$ and $\overline{P}_2$ is defined as follows:*

$$\overline{P} = \overline{P}_1 \|^a \overline{P}_2 = \langle V, \overline{\Sigma} \rangle, where \begin{cases} V = V_1 \cup V_2 \\ \overline{\Sigma} = \overline{\Sigma}_1 \wedge^a \overline{\Sigma}_2 \end{cases}$$

where, $\wedge^a$ denotes the conjunction of unclocked traces of a variable defined as follows:

$$\forall v \in (V_1 \cap V_2) : \overline{\sigma}(v) = \overline{\sigma}_1(v) = \overline{\sigma}_2(v)$$

$$\forall v \in (V_1 \setminus V_2) : \overline{\sigma}(v) = \overline{\sigma}_1(v)$$

$$\forall v \in (V_2 \setminus V_1) : \overline{\sigma}(v) = \overline{\sigma}_2(v)$$

The asynchronous composition ($\|^a$) operator is associative and commutative. For transforming a synchronous observation to an asynchronous observation, we define *declocking* as follows:

Definition 20. Declocking: $\langle V, \Sigma, \hat{c} \rangle \rightarrow \langle V, \overline{\Sigma} \rangle$. *It involves using the $asyntrace$ function, where $\forall v \in V, asyntrace : \Sigma \rightarrow \overline{\Sigma}$, which transforms $\sigma(v) \in \Sigma$ to $\overline{\sigma}(v) \in \overline{\Sigma}$. The clock associated with the variables is removed during this transformation. The $asyntrace$ function is defined as follows: $asyntrace(\sigma(v)) = un(\sigma(v), 1)$, where*

$$un(\sigma(v), i) = \begin{cases} un(\sigma(v), i+1), & \textbf{if } (\sigma(v)[i] = \tau) \\ \sigma(v)[i]@un(\sigma(v), i+1), & otherwise \end{cases}$$

Declocking removes all the absent events from the clocked traces of a synchronous observation. As a result, we get an unclocked trace for each clocked trace. Next, we define *reclocking*, such that given a clock, it transforms an asynchronous observation to a synchronous observation. *Reclocking* is defined as follows:

Definition 21. Reclocking: $\langle V, \overline{\Sigma} \rangle \times \mathbf{C} \rightarrow \langle V, \Sigma, \hat{c} \rangle$ *For any $\mathbf{c} \in \mathbf{C}$, $\forall v \in V$, we define $syntrace : \overline{\Sigma} \times \mathbf{c} \rightarrow \Sigma$ where*

$syntrace(\overline{\sigma}, \mathbf{c}) = f(\overline{\sigma}, \mathbf{c}, 1)$ *and*

$$f(\overline{\sigma}, \mathbf{c}, i) = \begin{cases} \overline{\sigma}[i]@f(\overline{\sigma}, \mathbf{c}, i+1), & \textbf{if } (t_i \in \hat{c}) \\ \tau@f(\overline{\sigma}, \mathbf{c}, i+1), & otherwise \end{cases}$$

The function $synctrace$ is applied to all unclock traces to transform them to clocked traces. The function places events of an unclock trace with respect to a clock provided, to form a clock trace. An absent event is placed when no event is present for a clock interval.

Until now, we have defined declocking to transform a synchronous observation to an asynchronous observation, and reclocking to transform an asynchronous observation back to a synchronous observation when a clock is provided. Now, we define synchronous composition of synchronous observation, where each observation may have a different clock. For a synchronous composition of two processes, one clock is selected for the resultant process and enforced for both the processes. Hence, declocking and reclocking has to be performed as shown below with the selected clock to synchronize them and obtain a synchronous observation.

Definition 22. Synchronous Composition of Synchronous Observations (||) *The synchronous composition of synchronous observations for processes, P_1 and P_2 is defined as follows:*

$$P = P_1 \parallel P_2 = \langle V, \Sigma, \hat{c} \rangle, where \begin{cases} V = V_1 \cup V_2 \\ \Sigma = \Sigma_1 \wedge^{\hat{c}} \Sigma_2 \\ \hat{c} = \hat{c}_1 \mid \hat{c}_2 \end{cases}$$

and $\wedge^{\hat{c}}$ denotes the conjunction of clocked traces of each variable. Since, all variables in a synchronous process have the same clock, we need to transform the traces of all variables from one process to another depending on whose clock is chosen. Considering P_1's clock is chosen ($\hat{c} = \hat{c}_1$). Then,

$$\forall v \in V_1 : \overline{\sigma}(v) = \overline{\sigma}_1(v)$$

$$\forall v \in (V_2) : \overline{\sigma}(v) = syntrace(asyntrace(\overline{\sigma}_2(v)), \hat{c}_1)$$

The synchronous composition operator (||) is commutative and associative. The communication between two observations happens on shared variables. Now, when these traces are composed together, the clock for the traces of the shared variable in one of the observations will be updated along with rest of the variables. Consider the following example:

Example 3. *We compose the $increment$ process (defined earlier) with an $isEven$ process, $\langle V, \Sigma, \hat{c} \rangle$, and $V = \{v_2, v_3\}$ and $\Sigma = \{\sigma(v_2), \sigma(v_3)\}$. The $isEven$ process is defined as follows: $v_2 = y$ and $v_3 = g(y) =$ $\textbf{if}\ (y\ mod\ 2\ =\ 0)\ \textbf{then}\ true\ \textbf{else}\ false$. The composed process will have three variables with v_2 as the common variable. The trace of v_2 is $\sigma(v_2) = e_{21}\ \tau_2\ \tau_3\ e_{24}\ \tau_5\ \tau_6\ e_{27}$ with e_{21}, e_{24}, e_{27} being $(6, t_1), (8, t_4), (9, t_7)$ and v_3 is $\sigma(v_3) = e_{31}\ \tau_2\ \tau_3\ e_{34}\ \tau_5\ \tau_6\ e_{37}$ with its events e_{31}, e_{34}, e_{37} as $(true, t_1), (true, t_4), (false, t_7)$. Now, assuming the clock of the $increment$ process is chosen to be the clock of the composed process, we declock and reclock the traces of $isEven$ process giving $\sigma(v_2) = e_{21}\ \tau_2\ e_{23}\tau_4\ e_{25}$ with e_{21}, e_{23}, e_{25} as $(6, t_1), (8, t_3), (9, t_5)$ and $\sigma(v_3) = e_{31}\ \tau_2, e_{33}\ \tau_4\ e_{35}$ with its events e_{31}, e_{33}, e_{35} as $(true, t_1), (true, t_3), (false, t_5)$. The clock of the composed process is same as the clock of $increment$ process.*

We have presented our trace based framework, where both asynchronous as well as synchronous aspects of an SoC can be modeled. In the next section, we define various notions and characterizations needed for correct validation of SoCs with GALS, where IPs are composed asynchronously.

7.4 Latency Equivalence for GALS

In this section, we show that IPs composed asynchronously are functionally equivalent to IPs composed synchronously given our characterization. Our correctness criterion is based on the notion of latency equivalence, which is defined as follows:

Definition 23. Latency Equivalent ($\equiv_e$): *Two traces, σ_1 and σ_2, are said to be* latency equivalent *iff $asyntrace(\sigma_1) = asyntrace(\sigma_2)$.*

This notion can be extended to processes. Two processes are said to be latency equivalent iff their outputs are latency equivalent when both are provided with the same inputs. In our framework, we describe a process in terms of its observations.

Lemma 1. *A synchronous observation of a synchronous process is latency equivalent to an asynchronous observation of the same process.*

Proof sketch: For a synchronous observation, the order of informative events for each clocked trace remain the same after the $asyntrace$ function is applied to it. This implies that two observations are latency equivalent. In other words, a synchronous observation and its declocking (i.e. its asynchronous observation) are latency equivalent.

We now define the notion of correctly declocking a synchronous process. We say that a synchronous process can be correctly declocked if the following condition holds:

Consider two clocked processes P_1, P_2 and their corresponding unclocked processes $\overline{P_1}, \overline{P_2}$, then

$$\textit{Condition 1: } P_1 = P_2 \Leftrightarrow \overline{P_1} = \overline{P_2}$$

Condition 1 states that given two clocked processes, if their traces in the clocked domain are equal then they can be uniquely declocked and their traces in the unclocked domain will be equal. Also, given two processes in the unclocked domain, they can be transformed to clocked processes uniquely given a clock, and therefore will have the same traces. Before we check the condition for declocking, we first define the *single activation property*:

Property 5. Single Activation*: For a clocked process, $\forall v \in V, \hat{v}_i = \hat{v}_j$.*

The single activation property means all the variables in a synchronous process are updated simultaneously. For the associated clock, the absence of events occurs at the same timestamps for all variables.

We now prove that if property 1 holds, then condition 1 is true. We consider two cases:
Case 1 *Declocking* $P_1 = P_2 \Rightarrow \overline{P_1} = \overline{P_2}$: If L.H.S (synchronous) is equal then R.H.S. (asynchronous) is also equal. The proof of this condition is trivial due to the *single activation property*. The removal of clock from both processes, P_1 and P_2, results in removal of absent events from the clocked traces of P_1 and P_2, makes them equal in the unclocked domain.
Case 2 *Reclocking* $P_1 = P_2 \Leftarrow \overline{P_1} = \overline{P_2}$: Here, we consider the reverse that if two processes, $\overline{P}_1$ and $\overline{P}_2$, are equal in the unclocked domain, then their synchronous processes are also equal. We use our reclocking function to recreate the clocked trace. For any $\mathbf{c} \in \mathbf{C}$, $\forall v \in V$, $synctrace$ is applied. Application of $syntrace$ to $\overline{P}_1$ and $\overline{P}_2$ results in insertion of events for a variable at the same timestamp when the clock is present. Hence, for both processes created P_1 and P_2, the number of events inserted and the timestamp of insertion are the same. Therefore, $P_1 = P_2$ if $\overline{P}_1 = \overline{P}_2$. Hence, condition 1 satisfies for synchronous processes.

Therefore, we can conclude that for any synchronous process, its corresponding asynchronous observation can be correctly formed. Implication: For a $\sigma \in \Sigma$ there exists a unique $\overline{\sigma} \in \overline{\Sigma}$. We conclude that given this transformation, an observation of any synchronous process can be correctly seen in an unclocked domain. By correctness here, we imply that the traces in the clocked and the unclocked domain are latency equivalent.

We have shown that given two processes with single activation property, the asynchronous observations and synchronous observations are latency equivalent. This implies that the asynchronous composition of such processes will also be latency equivalent to their synchronous composition. This follows from the definition of synchronous and asynchronous compositions. Therefore, a simple barrier synchronization protocol is sufficient for their asynchronous composition. Hence, we formulate the following theorem.

A system (Φ) is a composition of observations of processes, where the composition is commutative and associative. A synchronous system is a synchronous composition of synchronous processes, whereas an asynchronous system is a asynchronous composition of asynchronous components. Next, we look at declocking a synchronous system.

To correctly declock a synchronous system, it must have the property of single activation. Consider a synchronous system $\Phi = P_1 \parallel P_2$ and its declocking $\overline{\Phi} = \overline{P}_1 \parallel^a \overline{P}_2$. The following conditions must satisfy:

1. **Declocking a synchronous system:** For an observation of a synchronous system Φ, its corresponding asynchronous observation, $\overline{\Phi}$, can be uniquely formed. To prove this, we will consider $P_1 = \langle V_1, \Sigma_1, \hat{c}\rangle$ and $P_2 = \langle V_2, \Sigma_2, \hat{c}\rangle$. (1) $\forall v_i \in V_1$, a unique $\overline{\sigma}(v_i)$ can be formed from $\sigma(v_i)$. This follows from Lemma 1 and the definition of declocking. (2) Similarly, $\forall v_j \in V_2$, a unique $\overline{\sigma}(v_j)$ can be formed from $\sigma(v_j)$. (3) $\forall v \in V_1 \cap V_2, \overline{\sigma}(v)$ can be formed uniquely. To prove this, we know that for a synchronous composition, $\forall v_i, v_j \in V$ if $\sigma(v_i) = \overline{\sigma}^*$ and $\sigma(v_j) = \overline{\sigma}^*$, then $v_i = v_j$. Now, we consider $v_x, v_y \in V_1 \cap V_2$ and $v_x \neq v_y$. Therefore, $\sigma(v_x) \neq \sigma(v_y)$. Since, the position of absent events in the two traces is the same, removal of absent events from the trace would yield $\overline{\sigma}(v_x) \neq \overline{\sigma}(v_y)$ as the events on both traces will be different. Therefore, $\forall v \in V_1 \cap V_2$, $\overline{\sigma}(v)$ will be unique. Hence, Φ can be uniquely declocked to $\overline{\Phi}$.

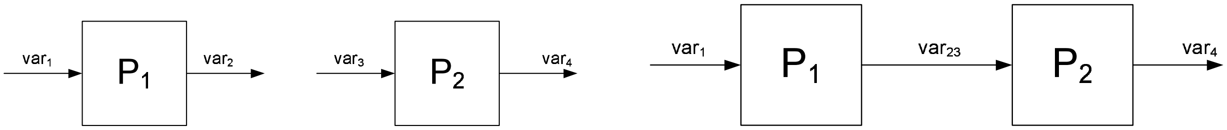
2. **Reclocking a synchronous system:** For a system $\overline{\Phi}$ given a clock **c**, its Φ can be uniquely formed. The proof of this is similar to the previous one for declocking. Considering the three cases: (1) $\forall v_i \in V_1$, a unique $\sigma(v_i)$ can be formed given **c** from $\overline{\sigma}(v_i)$. Give the clock **c**, a unique $\sigma(v_i)$ can be formed using the function $syntrace$. (2) Similarly, $\forall v_j \in V_2$, a unique $\sigma(v_j)$ can be formed from $\sigma(v_j)$ and clock **c**. (3) $\forall v \in V_1 \cap V_2$, $\sigma(v)$ can be formed uniquely. For $v_x, v_y \in V_1 \cap V_2$ and $v_x \neq v_y$, we get $\overline{\sigma}(v_x) \neq \overline{\sigma}(v_y)$. The timestamps at which the absent events are inserted is same for both traces. As the events for both traces are different, we get $\sigma(v_x) \neq \sigma(v_y)$ and hence are unique. Therefore, the asynchronous system can be reclocked uniquely given a clock.

Theorem 2. *If the single activation property holds for a system of IPs, then their asynchronous composition for GALS will be behaviorally equivalent to their synchronous composition.*

The proof of this theorem follows from previous discussions.

7.5 Analysis of RRCF designs

A *RRCF* system consists of components that are driven by related clocks. As discussed earlier, the clocks of all the processes of such systems are related at the root of the *clock tree*. The root of the clock tree is considered the fastest clock or fine-grained clock. All the other clocks are derived from this root clock.



(a) IP Blocks

(b) Composition of IP Blocks

Figure 7.1: IP Blocks and their composition

For example, consider two processes P1 and P2, and var_1 and var_2 are the input and output variables of P1, and var_3 and var_4 are the input and output variables of P2 (shown in Figure 7.1(a)). The traces of the two processes are shown below:

For process P1:

	t_1	t_2	t_3	t_4	t_5	t_6
var_1	e_1	e_2	e_3	e_4	e_5	e_6
var_2	f_1	f_2	f_3	f_4	f_5	f_6

For process P2:

	t_1’	t_2’	t_3’
var_3	f_1’	f_2’	f_3’
var_4	g_1’	g_2’	g_3’

Now, consider that the clock of process P1 is twice the clock of process P2. The composition of these blocks are shown in Figure 7.1(b). The processes are composed at var_2 and var_3, and we assume that the traces of both these variables are agreeable (as per definition of synchronous composition). We denote the composed trace by var_{23}. The corresponding traces of the composition are shown below.

	t_1	t_2	t_3	t_4	t_5	t_6
var_1	e_1	e_2	e_3	e_4	e_5	e_6
var_{23}	f_1	f_2	f_3	f_4	f_5	f_6
var_4	τ	g_1'	τ	g_2'	τ	g_3'

Now, if such a design were transformed to GALS, a special protocol for this design would be required to ensure that process P_2 only executes and produces valid results when f_2, f_4 and f_6 are present. In our framework, we attempt to transform this to a system with single activation.

In our framework, to compose these processes together, we first get the *clock tree* of the system. For this example, as process P1 has a clock twice faster than P2, the clock of P1 is considered to be the root clock of its clock tree. Now, composing the two processes together and providing the output of process P1 to the input of process P2 would mean that process P2 will only be able to process every alternate event from P1. We then try to map all the traces of the processes to the traces driven by the root clock of the system, which in this case is the clock of process P1. Therefore, the traces of process P2 will map to the traces from the clock of P1.

In other words we have to transform the traces of the slower process to the fastest one which is the root clock. For this, we define an *affine transformation* function that transforms traces based on one clock to another clock. Before, we define the transformation function, we introduce a new type of event called a *dummy event* of the form (X,t), where X denotes a *don't care* value, and t denotes its time stamp. This dummy event is denoted by a 'χ'. A dummy event is distinct from an absent event in the form that its value is a don't care instead of absent.

Now, we define *affine transformation* of the traces of one clock to the traces of a new clock. Affine clock calculus has been presented as an extension of formal verification techniques provided by SIGNAL language [203]. Our affine transformation notions are based on [204]. The affine transformation is dependent on three parameters: (n, φ, d), where $n, d \in \mathbb{N}^+$ and $\varphi \in Z$. Here, $n-1$ denotes the number of instants present between any two events in the current trace. For our case, as the current trace is rationally related to the root clock, n is always considered to be 1. In other words, passing of 1 instant in the current trace will denote passing of d time instants in the new trace which is based on the root clock. The φ^{th} instant denotes the starting instant in the new trace, and every d^{th} instant starting from the φ^{th} instant denotes the position where the informative events from the current trace are placed. For transformation of clocks of the processes, we know these three parameters. Affine transformation is functionally defined as follows:

Definition 24. *Affine transformation:* $\Sigma \times Z \times \mathbb{N}^+ \rightarrow \Sigma$. *For any* $\sigma \in \Sigma$, *given its new relation* d, *and initial starting point* φ, *we define* $\mathcal{AF}(\sigma, \varphi, d) = tr(\sigma, \varphi, d, 1, 1, \varphi)$, *where*

$$tr(\sigma,\varphi,d,i,j,k) = \begin{cases} \sigma[i]@tr(\sigma,\varphi,d,i+1,j+1,k+d), j=\varphi \vee j=k \\ \chi@tr(\sigma,\varphi,d,i,j+1,k), otherwise \end{cases}$$

Affine transformation transforms a trace in one clock domain to a trace in a different clock domain given the parameters (n,φ,d). Note that as n is always 1, we do not provide it as a parameter to the function. The variable $i, j,$ and k are used for this transformation where i denotes the index of the trace with the original clock, j denotes the index of the trace with the root clock, and k denotes the position at which the informative events are placed. The informative events are either placed when $j=\varphi$ which means that it is the position of the initial informative event with the new clock, or $j=k$ denotes when the k^{th} position is reached.

Now, considering the same example of the system with P1 and P2, and variables var_1, var_2, var_3 and var_4, we have n=1, φ=2 and d=2. Application of affine transformation to the traces of process P2 results in the following:

	t_1	t_2	t_3	t_4	t_5	t_6
var_3	χ_1	f_2	χ_3	f_4	χ_5	f_6
var_4	χ_1	g_2	χ_3	g_4	χ_5	g_6

We now want to enable communication on var_2 for P1 and var_3 for P2. This communication can only be allowed if the two variables have identical traces. We realized that the traces of the two variables are not identical because at timestamps t_1, t_3 and t_5, dummy events are seen on the trace of var_3, whereas informative events are seen on the trace of var_2. However, we realize that the instants at which the informative events of var_2 and var_3 are present are identical ($t_2, t_4,$ and t_6). Since, all the informative events on the traces of two variables are identical, we want the compose the two processes with these shared variables. To synchronously compose the two processes on these shared variables, we will need to update our definitions of conjunction $\wedge^c$ to include the dummy events. We now have $f_i \wedge^c \chi_i = f_i$ (Definition 22).

The composed trace of the system is as follows:

	t_1	t_2	t_3	t_4	t_5	t_6
var_1	e_1	e_2	e_3	e_4	e_5	e_6
var_{23}	f_1	f_2	f_3	f_4	f_5	f_6
var_4	χ_1	g_2	χ_3	g_4	χ_5	g_6

The trace at the output of the process (var_4) is our desired trace. Now, this system satisfies our notion of single-activation because at every instant there is a valid event, which is either in the form of informative event or a dummy event.

Given the *affine transformation* function and the parameters, the new trace based on the root clock is uniquely formed. We also present the inverse of the *affine transformation* to get the trace with its original clock.

Definition 25. *Inverse of Affine transformation:* $\Sigma \times Z \times \mathbb{N}^+ \rightarrow \Sigma$. *For any* $\sigma \in \Sigma$, *given its relation* d, *and initial starting point* φ, *we define* $\mathcal{AF}^{-1}(\sigma, \varphi, d) = itr(\sigma, \varphi, d, 1, 1, \varphi)$, *where*

$$itr(\sigma, \varphi, d, i, j, k) = \begin{cases} \sigma[i]@itr(\sigma, \varphi, d, i+1, j+1, k+d), j = \varphi \vee j = k \\ itr(\sigma, \varphi, d, i, j+1, k), otherwise \end{cases}$$

The $\mathcal{AF}^{-1}$ function transforms a trace with the root clock to the trace with its original clock. Here, i denotes the instant in the original clock, j denotes the instant in the root clock and k denotes the position where the informative events are placed. Therefore, we can uniquely transform a trace with its original clock to a trace with its root clock and reverse.

Now, we need to ensure that the insertion of dummy events do not change the semantics of the RRCF system. We know that in such a system, a process only reads the values based on its clock. When these processes are mapped to the root clock, the informative events are still seen on their original clocks, only the timescale is changed to the root clock. The dummy events and the absent events are seen only when the original clock of the process is absent. Moreover, the dummy events only replace the absent events which are as well ignored by the synchronous process. In real hardware, an absent event for a synchronous process means that the value is irrelevant for the process. Hence, these values are ignored for the synchronous system; however, in a GALS design, when we have a communication protocol, it would not see absent events, but would see valid events. Therefore, dummy events help ensure that the events read by the synchronous islands are synchronized. This helps in the case when a process has two inputs, but is only expecting value on one of the inputs. In our example, when we map var_3 and var_4 to the root clock, the informative events are only seen at every other timestamp because the clock of process P2 is half as fast as the root clock. Instead of inserting absent events we insert dummy events such that when the design is transformed to a GALS design, these dummy events are seen. Therefore, the semantics of P2 does not change. Now, when we transform this to our GALS protocol, the dummy events would be synchronized, when they are read from the barrier synchronized protocol. Therefore, even in the case when any of the inputs of a synchronous island are delayed, the protocol would stall the synchronous island until all the inputs are available. This would ensure the exact same behavior as in when the design is mapped to the root clock. Therefore, the barrier synchronized protocol for GALS would still be correct-by-construction as the dummy events would be seen by the barrier as valid events, however, the process in GALS will only accept informative inputs on its input signal based on its clock. The clock of a synchronous island in the GALS will only execute when valid inputs are seen. Also, for specific cases, the barrier synchronized GALS protocol can filter out the dummy events, and only provide informative events to the synchronous process.

7.6 GALS Protocol

In this section, we present a protocol for communication in a GALS design. Once we realize that a system has single activation, we then encapsulate each process (synchronous island) with our encapsulation scheme with input and output interface processes.

The protocol involves refinement of each component with: (1) Input interface process with barrier synchronization (IIP), (2) output interface process (OIP). Asynchronous FIFOs are placed between two components for communication. These FIFOs are equipped with interfaces that ensure correct communication between components with independent clocks. Detail information about these FIFOs can be found in [79]. Figure 7.2 illustrates the block diagram of a component in this architecture.

The block diagram of the process along with the protocol is shown in Figure 7.2.

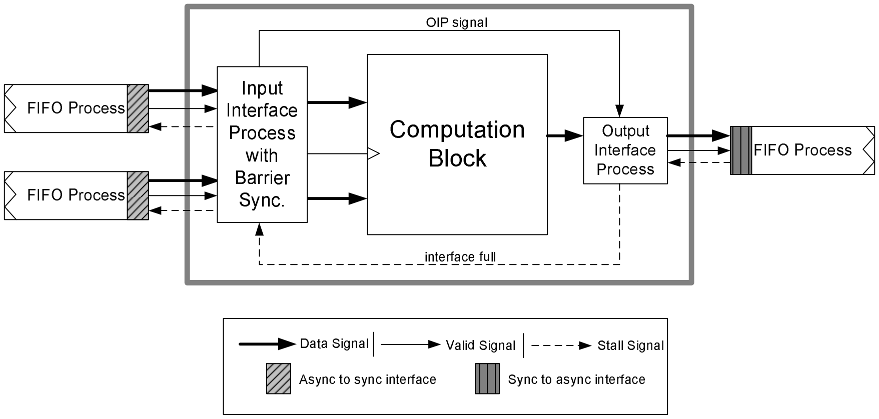


Figure 7.2: Block Diagram of the Protocol

In the GALS design, each signal is associated with a valid and a stall signal. The idea of valid-stall signal is borrowed from the existing works on LIPs [69, 70, 206, 72, 201, 89, 88]. A sender sends valid data (validity of data denoted by valid signals) to its receiver on every clock, and whenever the stall signal is not set. Once the stall is asserted, the sender does not send valid data.

Input interface process with barrier synchronization (IIP): The IIP is placed at the input of the synchronous component. The main idea of this process is to barrier synchronize (align) all the valid inputs for the computational block. The block can only execute once all the inputs have been realized. Each IIP contains buffers for each input signal to store input data values. There are exactly two storage elements for each input. This is because when the computation block is stopped by the IIP, the incoming inputs need to be stored, and the stall signals for the appropriate source components have to be enabled. The need for the stall signal is realized as soon as the first storage element is filled. By the time the stall signal is enabled, the source component could have placed another valid value on the signal. Therefore, the second storage is needed to store this value. For each input to the component, IIP has an input data signal, an input valid signal and an output stall signal. Furthermore, it has two signals to communicate with OIP: (1) OIP signal which tells the OIP that a valid value has been processed by the computation block, and (2) Interface full signal from the OIP which tells the IIP that the OIP is ready to read a value or not. The IIP works in two phases: In the first phase it reads all inputs and stores the data values in its buffer. In the second phase it provides the data values to the component based on its input valid signals which are written to its output.

There are three possible scenarios that can occur:

1. All input valid signals are 1: This mean that valid values are seen on all inputs. During the first phase, the IIP stores the data, and at the start of the second phase, the values are provided to the computation block. Also, the IIP outputs a 1 on its OIP signal. The output stall signals to its source components are disabled (i.e. 0 is placed on them). The clock for the computation block is enabled.

2. One of the input valid signals are 0: One of the input does not have valid data. In the first phase, the IIP reads all the valid values from its inputs, and stores them in their respective storage elements. During the second phase, the IIP sends 0 on its OIP signal. This causes OIP to realize that the computation block has not produced a valid data. For the inputs where valid value was not realized, the IIP places 0 on their corresponding stall signals. All other output stall signals are enabled (set as 1). Now, in the following cycle, if the IIP realizes the valid values on all the inputs where valid was not seen in the previous cycle, then the procedure in scenario 1 follows. Otherwise, the stall signals remain unchanged for the inputs where valid values are not seen.

3. Interface full signal is 1 (enabled): This means that the OIP is not able to accept a valid data from the computation block. In the first phase, the IIP will read and store the inputs. In the second phase, the computation block is stalled, and the IIP outputs a 0 on OIP. The stall signals are set based on scenario 2.

Output interface process (OIP): The OIP is placed at the output of the synchronous component, and contains one buffer to store the result of the synchronous block. The inputs of OIP include valid value from the computation block, OIP signal from the IIP, and a stall signal from its destination FIFO. OIP reads and stores the value from the computation block whenever a 1 is received from the IIP (second phase). The OIP places the valid value received from the computation block to its output when the stall signal from the FIFO is disabled. In the case when the stall signal from the FIFO is enabled, the data from the computation block is stored in its buffer, and 1 is placed on the interface full signal to the IIP.

FIFO process: The $FIFO$ process provides the communication between two synchronous components. At each end of the FIFO, there are interfaces that communicate with the synchronous component. Note that this FIFO stands as an interface between the components running on different clocks. So, this FIFO has an synchronous to asynchronous interface on its input end, and an asynchronous to synchronous interface on its output end. Details about such interfaces can be found in [79]. The FIFO enables a stall to its source component when the buffer becomes full. Valid data is written on its output based on the clock of the destination component when valid value is present.

7.7 Case Study: FIR Filter

The FIR filter system consists of three different modules: Stimuli, FIR and Display (Figure 7.3). We analyze the three IPs independently in our trace based framework. For all the three modules, we show the abstract behavior represented by variables. The Stimuli module generates valid data. This module initially takes four cycles for resetting, and data is generated every ten cycles. For Stimuli module, we represent its behavior by three variables: *trigger*, *Sticycle* and *datastart*. This FIR model is based on the SystemC model described and distributed with SystemC distribution [13].

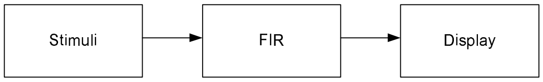


Figure 7.3: FIR filter

The variable *trigger* is a boolean (T or F) type, and is used to trigger the Stimuli module. This acts as the clock for the stimuli module. The variable *Sticycle* is used to illustrate the reset cycles, and finally the variable *datastart* is the data output. When a trigger is received, the *Sticycle* starts, and at the fourth cycle, the valid data, denoted by $vdata$ is seen on the variable *datastart*. The trace of the variables of the Stimuli IP is shown below:

$$\begin{array}{rcccc} & t_1 & t_2 & t_3 & t_4 \\ \sigma(trigger): & T & \tau & \tau & \tau \\ \sigma(Sticycle): & 1 & 2 & 3 & 4 \\ \sigma(datastart): & \tau & \tau & \tau & vdata \end{array}$$

It can be realized that the module does not have the single activation property, since the clocks of variables $trigger$, $Sticycle$ and $datastart$ are not the same. As this module does not have single activation, we cannot declock this module in our framework. We now assume that all the modules possess single activation. Therefore, to make this module single activation, we consider all variables of an IP have a valid value at time stamp whenever any variable has a valid value. Therefore, for the variable $trigger$, we consider false (F) instead of τs to ensure that it has a valid value which is false. Also for the variable $datastart$, we consider 0 to be an invalid value, but the value is not absent. The invalid values are discarded in implementation. The new trace is shown below:

$$\begin{array}{rcccc} & t_1 & t_2 & t_3 & t_4 \\ \sigma(trigger): & T & F & F & F \\ \sigma(Sticycle): & 1 & 2 & 3 & 4 \\ \sigma(datastart): & 0 & 0 & 0 & vdata \end{array}$$

Next, we consider the FIR module where the computation occurs in ten cycles. We consider three variables for the FIR module: *datastart, FIRcycle* and *compdata*. The *datastart* variable either has valid or invalid data (which is 0), or the data is absent. The *FIRcycle* variable represents the computation cycles. We abstract out the computation of the FIR, and represent only its computational cycle to make the example easy to understand. The *compdata* variable has a valid value when the computation is complete, denoted by $cdata$, otherwise it is 0 or absent. The valid $cdata$ can be seen on variable $compdata$ ten cycles after $vdata$ is seen on variable $datastart$. The trace for the FIR module is shown below:

$$\begin{array}{rccccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & \dots & t_{14} \\ \sigma(datastart): & 0 & 0 & 0 & vdata & 0 & \dots & 0 \\ \sigma(FIRcycle): & 1 & 2 & 3 & 4 & 5 & \dots & 14 \\ \sigma(compdata): & 0 & 0 & 0 & 0 & 0 & \dots & cdata \end{array}$$

Similarly, we can denote the traces of the variable of the display module.

Now, we consider their compositions. For the Stimuli and FIR module, we realize that the common variable *datastart* has the same trace. Therefore, the two modules are composed and communication happens on this shared variable. Since, both these modules have the property of single activation, their synchronous composition also has single activation[4]. Similarly, the Display module is composed with Stimuli and FIR module. We get a design of these three modules interaction, which also has the property of single activation. Given this property, from theorem 2, this system can be correctly implemented as a GALS system. The traces of the variables and there composition aids in validating the design. Also, due to the availability of various verification tools for synchronous designs, we can formally verify the synchronous version of this SoC, and its correct GALS implementation can be implemented without the need for verifying this GALS design.

[4] Follows from the definition of synchronous composition

Chapter 8

Refinement of GALS Architectures from KPN

In Kahn process network (KPN), the processes (nodes) communicate by unbounded unidirectional FIFO channels (arcs), with the property of non-blocking writes and blocking reads on the channels. KPN provides a semantic model of computation (MoC), where a computation can be expressed as a set of asynchronously communicating processes. Therefore, if the input/output behaviors of each component are continuous, then it is an ideal MoC for specifying globally asynchronous locally synchronous (GALS) designs. However, the unbounded FIFO based asynchrony is not realizable in practice, and hence requires refinement when implemented in real hardware. In this chapter, our objective is to provide a correct-by-construction design methodology for creating GALS designs starting from a design specified as a KPN. To ensure a correct-by-construction GALS implementation, we need (i) to ensure that the refinement preserves the Kahn principle, where each refined process is continuous and monotonic, and (ii) to establish behavioral equivalence between the original Kahn specification and the refined GALS implementation.

To achieve this objective, we present a formal refinement based framework which takes us from a KPN specification towards a GALS implementation. We formally show that the GALS implementation preserves the Kahn property. Our formal methodology of refinement from a KPN specification into a GALS implementation is generic, and various alternate schemes of a GALS implementation can be derived. We consider four different GALS architectures, and provide the formal definitions for each of these. The architectures we consider involve different communication protocols. In the *handshake-based architecture*, the communication between processes occur based on well known four-phase handshake protocol, whereas in the *fifo-based architecture*, the communication protocol borrows ideas from latency insensitive protocols with a fifo interface connecting the processes running on different clocks. The *controller-based architecture* consists of a centralized controller that governs the execution of the processes, similar to dynamic tagged-token dataflow architecture [35]. Finally, we have the *lookup-based architecture* that avoids the signaling based protocols, and the communication happens based on fast data accesses from a lookup storage which is located on-chip.

The transformation to a GALS model from a KPN model involves: (i) replacing the unbounded storage elements (infinite FIFOs) with bounded storage elements, and (ii) enforcing blocking read and blocking write conditions on the process. The architectures we present, consist of processes that are continuous and monotonic, and are also correct-by-construction. We show this by transforming the processes into deterministic (I/O) automata, since they have been shown to be continuous and monotonic [156]. We show behavioral equivalence of the KPN and the refined architecture by ensuring that the behaviors of both are latency equivalent.

8.0.1 Design Methodology for GALS

The design methodology we propose for GALS design can be summarized as follows: Given the description of a system, the first step is to identify the behaviors of the system as a collection of concurrent processes communicating asynchronously with a Kahn Process Network (KPN) MoC. To ensure the correctness of the KPN model, the specification can validated via functional simulation. Architectural exploration can then be performed to identify the appropriate GALS architecture based on the KPN model. Based on the result of architectural exploration, appropriate refinements are applied to the KPN model. Figure 8.1 illustrates this design methodology.

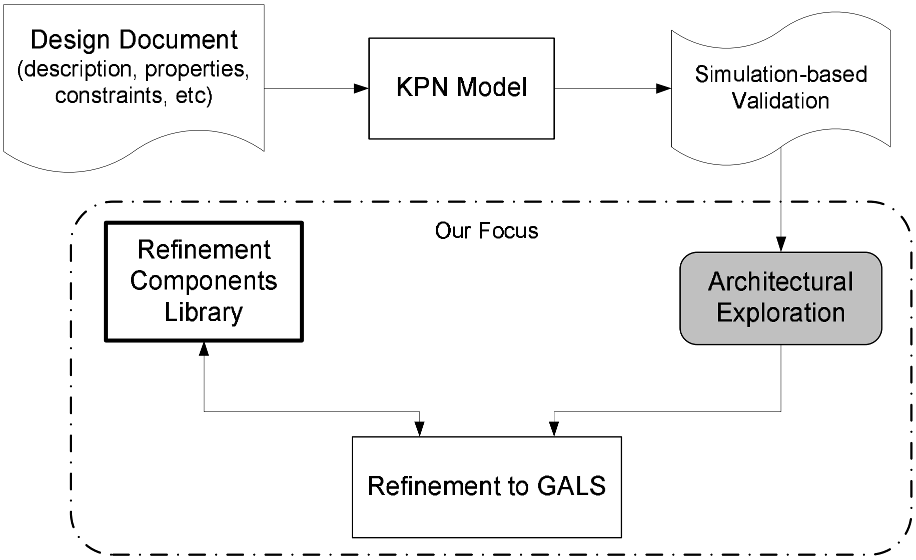

Design Document
(description, properties, constraints, etc)
KPN Model
Simulation-based Validation
Our Focus
Refinement Components Library
Architectural Exploration
Refinement to GALS

Figure 8.1: Design Methodology for GALS

8.0.2 Model of Computation for GALS

Most of the design languages used in the industry such as SystemC [13], SpecC [113], support synchronous specifications, and use discrete event as their MoC. A GALS design can be modeled in these design languages using discrete event as a MoC, however, it is not very natural. KPNs are well known simplistic models used for expressing behaviors involving dataflow such as streaming audio, video, and other 3D multimedia applications. The behaviors of the applications is modeled as a collection of concurrent processes communicating data via first-in-first-out (FIFO) channels with unbounded capacity. The processes in a KPN are sequential programs that consume data (referred as *tokens*) from their input channels and produce data on their output channels. These processes execute based on blocking read and non-blocking write schemes.

A KPN model closely relates to a GALS description, where the processes can be seen as synchronous components, and their interaction is asynchronous. There are various advantages of using a KPN as a model of computation for GALS: (1) The processes in a KPN are continuous which implies they are monotonic [138]. This means that a process can incrementally produce outputs when sufficient inputs are realized. This property applies well to the components of GALS designs where each component produces outputs when sufficient data is seen on its inputs. (2) The processes in KPN are determinate [138], and we want this property to hold true for the components in GALS. (3) The processes of a KPN are compositional which follows from them being determinate. To gain deterministic behavior in process networks with feedbacks, KPNs are also equipped with least fixed point semantics. This compositional property helps in realizing larger behaviors by composing smaller ones together. This is useful for designing GALS, where the components can be reused and composed together to specify large behaviors. (4) There is no notion of global data in a KPN which helps in safely implementing concurrency. The components of the GALS design execute concurrently. (5) KPN provides a formal semantic model to construct rigorous proofs for the properties of the computation aspect of the network of processes. The semantic model of KPN allows reasoning about the correctness of the design abstracting away the communication aspect. In KPN, the communication is assumed to be correct due to the assumption of infinite FIFO's with no data loss. In realistic refinement such unbounded communication is replaced by protocols and architectural artifacts, and hence the refinement needs to be proven as a formal refinement of the original KPN.

8.1 Preliminary Definitions

In this section, we discuss some of the definitions we use in the chapter. Let D be the set of data values, and T be a set of tags, where $T = \mathbb{N}$. The set of all events is denoted by E where, an event $e \in D \times T$ is defined as a value-tag pair. However, in the systems we consider, a special event called *absent event* denoted by τ may occur[1]. Hence, an event with a informative value is called an

[1] An absent event may occur due to lack of data in the producer or due to a consumer's request to delay a transmission.

informative event ($e \in D \times T$) otherwise it is an absent event (τ). There are many ways to indicate validity of events, for example using extra 'valid' signal with each data signal.

Definition 26. Signal: *A signal $s = e_i e_j e_k ...$ is defined as a sequence of events which are ordered based on their tags, where $i < j < k$ and $i, j, k \in \mathbb{N}$.*

For a signal s, $s[i]$ denotes its i^{th} event. The set of all signals is denoted by S. An empty signal is denoted by [ ].

Definition 27. *A computation process $\mathcal{P}$ is a function that maps a set of input signals to a set of output signals. $\mathcal{P} : S^m \rightarrow S^n$ with $m, n \in \mathbb{N}$.*

We now define helper functions that we use in the chapter.

Definition 28. *exp_1^n is a textual replacement of $exp_1, exp_2, ..., exp_n$.*

For example, signals $s_1, s_2, ..., s_n$ can be denoted as s_1^n. Abusing the notation, we extend this to the scope of the functions. A function $func$ used as $func(s_1^n)$ implies that all $s_i \in s_1^n$ from 1 to n are arguments to the function. On the other hand, a function used as $func(s)_1^n$ implies that the function is applied to each individual s_i as a single argument. Throughout the chapter, we use these short hand notations.

Definition 29. *Given $s \in S$ and $e \in E$, we define the prepend operator $\oplus$, where $e \oplus s = s'$, s.t. e is the first event and s is the rest of the signal.*

Definition 30. *Given a tuple of m signals and another of n signals, $\bigodot$ creates a tuple of $m + n$ signals.*

$$< a_1, \ldots, a_n > \bigodot < b_1, \ldots, b_m >=< a_1, \ldots, a_n, b_1, \ldots, b_m >$$

Definition 31. *Given two tuples of n events and n signals respectively, $\bigoplus$ creates a tuple of n signals with an event prepended each signal.*

$$< e_1, \ldots, e_n > \bigoplus < s_1, \ldots, s_n > = < e_1 \oplus s_1, \ldots, e_n \oplus s_n >$$

8.2 Formal Definition of Kahn Process Network (KPN)

A KPN consists of processes that communicate via point-to-point unbounded FIFO channels. Each process reads from **all** FIFOs on its inputs and writes to **all** FIFOs on its outputs. Consider $\mathcal{P}$ to be the set of all processes of the KPN, and each $p \in \mathcal{P}$ follows the blocking read and non-blocking write policies. A process executes when it is waiting on its inputs and the inputs arrive. A FIFO can be defined as a list, where the data is added to the end of the list and removed from the head of the list. The set of all FIFOs is defined by $\mathcal{F}$. Similar to the signals, we use the notation [ ] to define an empty fifo.

We define the functions src and dst to identify the source and destination processes of a FIFO. For a given FIFO, the function $src : \mathcal{F} \rightarrow \mathcal{P}$ returns the process that writes into the FIFO, and the function $dst : \mathcal{F} \rightarrow \mathcal{P}$ returns the process that reads from it. For example, if $p_1, p_2 \in \mathcal{P}$ and $f_1 \in \mathcal{F}$ is the FIFO channel connecting from p_1 to p_2, then $src(f_1) = p_1$ and $dst(f_1) = p_2$.

Table 8.1 defines some basic functions used in the chapter. We assume that x denotes a list elements, and d denotes a single element.

Function Name	Description
$rdF(x)$	Function rdF returns the element at the head of the list x.
$len(x)$	Function len returns the length of the list x.
$dequeue(x)$	Function $dequeue$ returns the list x without its head.
$enqueue(x, d)$	Function $enqueue$ returns the list x with d appended to its end.

Table 8.1: Basic Definitions

We now define the function to execute a process in KPN.

Definition 32. *The function $exP : \mathcal{P} \rightarrow \mathcal{F}^n \rightarrow \mathcal{F}^m$ executes the process $p \in \mathcal{P}$, with data on input FIFOs $fi_1^n \in \mathcal{F}$, and outputs the result on each of its output FIFOs $fo_1^m \in \mathcal{F}$. The following definition is of the output data that is enqueued to each output FIFO.*

$$exP(p) = \begin{cases} p(rdF((fi)_1^n)), & if\ \forall fi \in fi_1^n : len(fi) > 0 \\ [\,], & otherwise \end{cases}$$

Every time a process p is executed, $\forall f \in \mathcal{F} : dst(f) = p$, a $dequeue$ operation is performed. Since a process requires data on all its inputs, after its execution, all input fifos are dequeued once. Also, $\forall f \in \mathcal{F} : src(f) = p$, $enqueue$ operation on all output FIFOs is performed with the data produced by the process. In the case when source and destination processes for a FIFO execute concurrently, the $dequeue$ operation precedes the $enqueue$ operation. The following function defines the update on FIFOs.

Definition 33. *Consider the function $upF : \mathcal{F}^m \rightarrow \mathcal{F}^m$, where $fi_1^m \in \mathcal{F}$ be the FIFOs that are inputs, and returns the corresponding updated FIFOs. We assume that d_1^m are the data values produced when the source processes of the corresponding FIFOs execute. For each $f_i \in f_1^m$ and its corresponding $d_i \in d_1^m$, the function udt is called, and is defined as follows:*

$$udt(f_i) = \begin{cases} enqueue(dequeue(f_i), d_i), & if\ src(f_i) \wedge dst(f_i)\ execute \\ dequeue(f_i), & if\ dst(f_i)\ executes\ only \\ enqueue(f_i, d_i), & if\ src(f_i)\ executes\ only \\ f_i, & otherwise \end{cases}$$

Definition 34. *The process $kpn : \mathcal{P}^m \rightarrow \mathcal{F}^n \rightarrow \mathcal{F}^o$ defines the KPN network with m processes, n FIFOs, and o output FIFOs. These output FIFOs represent the signals that are not input to any processes in the network. Consider $p_1^m \in \mathcal{P}$ be the processes of the system, $fi_1^n \in \mathcal{F}$ be the FIFOs connecting these processes, and $fo_1^o \in \mathcal{F}$ be the FIFOs at the output of the KPN. The function kpn is defined as follows:*

$$kpn(p_1^m, fi_1^n, fo_1^o) = \begin{cases} fo_1^o, & if\ \forall fi \in fi_i^n : len(fi) = 0 \\ kpn(exP(p)_1^m, upF(fi_1^n), upF(fo_1^o)), & otherwise \end{cases}$$

Note that in the KPN model, there are no rules governing the execution of the processes, and can execute in any order.

In the following sections, we present refinement schemes to different architectures. For each of these architectures, we start with a KPN process, and compose it with the corresponding refinement components. The KPN process has inputs and outputs connected by the refinement components instead of unbounded FIFOs.

8.3 A Running Example

We use a running example to illustrate the functionality of different architectures and compare their pros and cons. We use the KPN diagram shown in Figure 8.2 that consists of four processes: $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ which connected by channels s_1, s_2, s_3, and s_4 with unbounded FIFOs. The processes $\mathcal{A}$ and $\mathcal{B}$ are source processes that produce tokens on channels s_1 and s_2. The process $\mathcal{C}$ has an initial token on channel s_4. A possible behavior of the network is as follows: Processes $\mathcal{A}$ and $\mathcal{B}$ execute producing tokens on their respective outputs. Process $\mathcal{D}$ cannot execute as there are no tokens from $\mathcal{C}$, so $\mathcal{C}$ executes first then followed by $\mathcal{D}$.

The components in GALS are associated with clocks, which are unknown to the designer. The clocks for these components are assumed to be independent. These clocks can either be generated locally by using gates such as inverters in a locked loop fashion, or can be from an external source.

For our KPN example of Figure 8.2, we consider sample clock ticks shown in Table 8.2. These ticks can be seen by an observer that is observing the design synchronously and analyzing the clock realization for different components. A clock tick represents a time stamp based on when the components are fired. The clock ticks (✓) signify when the clocks of the respective components are triggered. For example, the component $\mathcal{A}$ is observed to trigger at t_1, t_2, t_3, t_4, and t_6, and component $\mathcal{B}$ triggers at t_1, t_3, t_4, t_5, and t_6. From the clock table, it can be said that component $\mathcal{A}$ and $\mathcal{B}$ execute in parallel at t_1. Please note that the information presented in Table 8.2 is a sample observation from an observer when the design executes. These clock relationships are not known to the designers at design time. So, as far as the designer is concerned, the components are completely asynchronous with respect to each other.

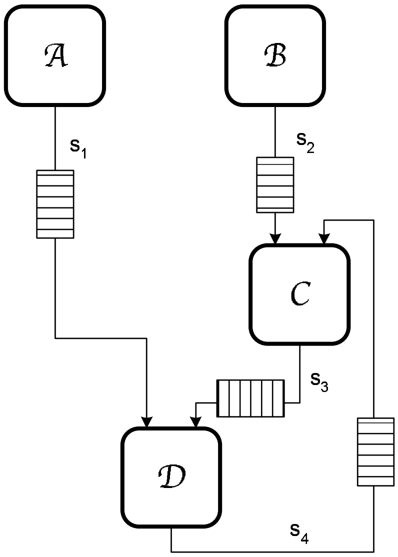


Figure 8.2: KPN model

ticks	t_1	t_2	t_3	t_4	t_5	t_6
$\mathcal{A}$	✓	✓	✓	✓	-	✓
$\mathcal{B}$	✓	-	✓	✓	✓	✓
$\mathcal{C}$	-	✓	-	✓	-	-
$\mathcal{D}$	-	✓	✓	-	✓	✓

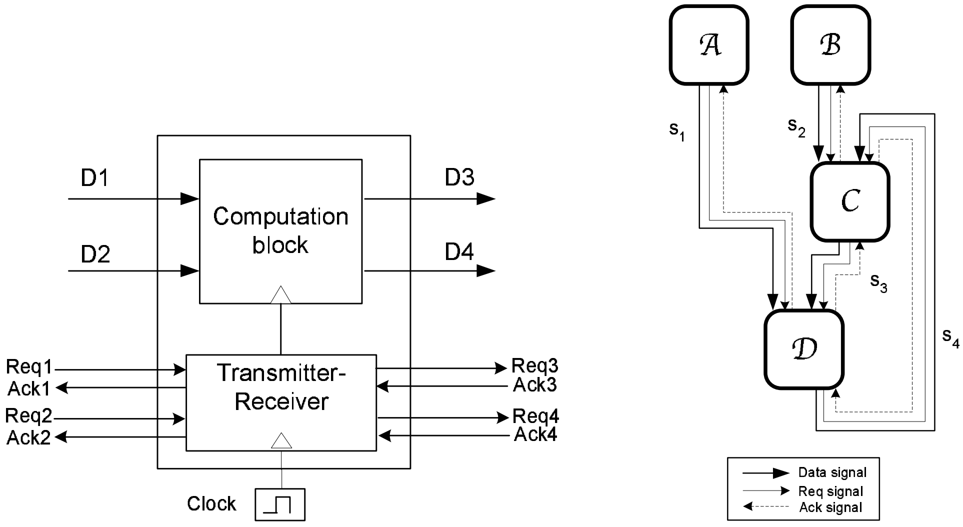
Table 8.2: Clock table corresponding to processes in Figure 8.2

8.4 Handshake based GALS Architecture

In the handshake-based GALS architecture, the synchronous components communicate directly via handshaking schemes. A receiver-transmitter unit (RTU) is added to each component to ensure proper request-acknowledge based handshake protocol. Each signal (carrying valid data) is augmented with two extra signals for control purposes: request and acknowledge. The components follow the signaling protocol discussed in Chapter 3.

Consider a source component and a destination component, where the source component sends data to the destination component. The data can be sent or received when it is triggered by its clock. Figure 8.3(a) shows a component with two input and two output signals. D1, D2, D3, and D4 are the input/output data signals, and req1, ack1, req2, ack2, req3, ack3, req4, and ack4 are its corresponding request and acknowledgement signals.

In a network, a synchronous component executes when the following conditions hold: (i) All its input request signals are requesting, (ii) all its input acknowledge signals are waiting for new request. Once, both these conditions hold, the component executes based on its clock. Until these



(a) Component with RTU (b) GALS example

Figure 8.3: Handshake-based GALS Architecture

conditions are true, the synchronous component is disabled.

Figure 8.3(b) can be refined from Figure 8.2 where the RTU are added to each process, and the communication between nodes handled by: data signal, request signal (req) and acknowledge signal (ack). Given handshake based GALS in Figure 8.3(b) and its corresponding clocks in Table 8.2,

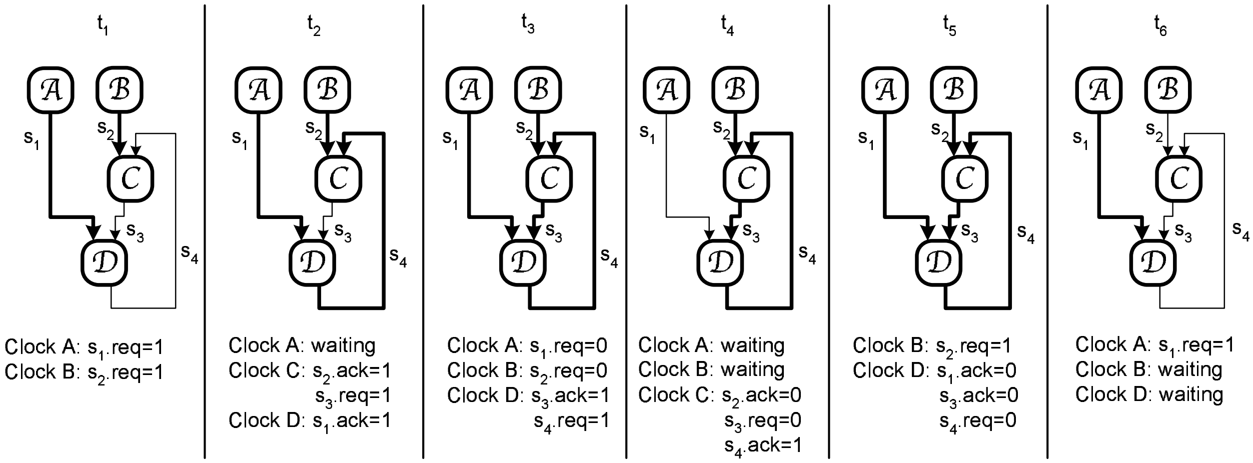


Figure 8.4: Simulation trace for handshake based architecture

we analyze its simulation trace based on four-phase handshaking protocol. Figure 8.4 illustrates what signals are updated at different clock ticks. Now, based on the clock table, components $\mathcal{A}$ and $\mathcal{B}$ trigger at t_1, i.e. $\mathcal{A}$'s $s_1.req$=1 and $\mathcal{B}$'s $s_2.req$=1. In t_2, component $\mathcal{C}$'s $s_2.ack$=1 and $\mathcal{D}$'s $s_1.ack$=1, however, component $\mathcal{A}$ keeps waiting as no acknowledgement has been received from component $\mathcal{D}$. Note that the data is transmitted when the request signal is set to '1', and the sender knows that the receiver is ready to receive a new value when acknowledge signal is set to '0'.

8.4.1 Formal Definitions of Handshake-based Architectures

In this section, we present formal semantics of the encapsulation that facilitates a four-phase handshake protocol.

Let the set of all data signals be denoted by S_d, request signals by S_{req} and acknowledge signals by S_{ack}, where $S_d \cup S_{req} \cup S_{ack} = S$. A data signal $d \in S_d$ is of the form $d = d_1 d_2 d_3 ...$, where $d_i \in E$. A request signal $r \in S_{req}$ is of the form $r = r_1 r_2 r_3 ...$, where $r_i \in \{0, 1\} \times \mathbb{N}$. Similarly, an acknowledge signal $a \in S_{req}$ is of the form $a = a_1 a_2 a_3 ...$, where $a_i \in \{0, 1\} \times \mathbb{N}$. The request and acknowledge signals are produced and consumed by the processes. Starting with process p, each process is refined with a receiver-transmitter unit that ensures communication with these request-acknowledge signals.

A receiver consists of a two-state machine with states $\{idle, wait\}$, while a transmitter consists of a three state machine with states $\{idle, req, wait\}$. The initial states for both receiver and transmitter machines are $idle$ for a process. The receiver sends a ready signal $rdy = 1$ (internal signal) to the transmitter (of the same process) to indicate that the process is ready to consume data. Similarly, the transmitter sends a ready signal $tdy = 1$ to indicate that the process is ready to produce data.

The process encapsulation is defined as follows:

Definition 35. *Consider the function $hba : S_d^n \times S_{req}^n \times S_{ack}^m \rightarrow S_d^m \times S_{req}^m \times S_{ack}^n$. Lets denote di_1^n : n input data signals; ri_1^n : n input request signals, and ai_1^m : m input acknowledge signals. Each encapsulated process contains $p \in \mathcal{P}$, and has receiver state r_{st}, transmitter state t_{st} with initial states as $idle$, and internal transmitter-receiver signals, $rdy = 0$ and $tdy = 1$. All other signals are initially set to 0. We use $c \in \mathbb{N}$ to denote the current index of the signals. The output data, request and acknowledge signals are denoted as do_1^m, ro_1^m, ao_1^n[2].*

The definition of the process hba is as follows:

$$hba(di_1^n, ri_1^n, ai_1^m) = encap(p, di_1^n, ri_1^n, ai_1^m, 0, idle, idle, 1, 0)$$

where, $encap(p, di_1^n, ri_1^n, ai_1^m, c, r_{st}, t_{st}, rdy, tdy)$ =

$$\begin{cases} do_1^m \bigodot ro_1^m \bigodot ao_1^n, & if\ \forall r \in S_{req} : r = 0 \wedge \forall a \in S_{ack} : a = 0 \\ do_1^m \bigodot ro_1^m \bigodot ao_1^n \bigoplus proc(p, di_1^n, ri_1^n, ai_1^m, c+1, r'_{st}, t'_{st}, rdy', tdy'), & otherwise \end{cases}$$

[2] Throughout the definitions, to show the current value and next value, we use the prime (') notation. For example, if x is the current value, then x' represents the next value.

where $do_1'^m, ro_1'^m, ao_1'^n, r'_{st}, t'_{st}, rdy'$, and tdy' are computed as shown below. For signals, $do_1'^m, ro_1'^m$, and $ao_1'^n$ the next values are shown by the definitions, which can be appended to the respective signals.

$do_1'^m$ is defined by:

$$For\ each\ d \in do_1^m, d' = \begin{cases} p(di_1^n), & if(r_{st} = idle \wedge t_{st} = idle \wedge \forall r \in ri_1^n : r = 1) \\ d, & otherwise \end{cases}$$

Note that the functionality of the process is defined by process p.

$ro_1'^m$ is defined by:

$$For\ each\ r \in ro_1^m,\ r' = \begin{cases} 0, & if(t_{st} = idle \wedge rdy = 1) \\ 1, & if(\forall a \in ai_1^m : a = 1 \wedge t_{st} = req) \\ r, & otherwise \end{cases}$$

$ao_1'^n$ is defined by:

$$For\ each\ a \in ao_1^n, a' = \begin{cases} 1, & if(\forall r \in ri_1^n : r = 1 \wedge r_{st} = idle \wedge tdy = 1) \\ 0, & if(\forall r \in ri_1^n : ri_1^n : r = 0 \wedge r_{st} = wait) \\ a, & otherwise \end{cases}$$

r'_{st} is defined by:

$$r'_{st} = \begin{cases} wait, & if(\forall r \in ri_1^n : r = 1 \wedge r_{st} = idle \wedge tdy = 1) \\ idle, & if(\forall r \in ri_1^n : r = 0 \wedge r_{st} = wait) \\ r_{st}, & otherwise \end{cases}$$

t'_{st} is defined by:

$$t'_{st} = \begin{cases} req, & if(rdy = 1 \wedge t_{st} = idle) \\ wait, & if(\forall a \in ai_1^n : a = 1 \wedge t_{st} = req) \\ idle, & if(\forall a \in ai_1^n : a = 0 \wedge t_{st} = wait) \\ t_{st}, & otherwise \end{cases}$$

rdy' is defined by:

$$rdy' = \begin{cases} 1, & if(\forall r \in ri_1^n : r = 1 \wedge r_{st} = idle \wedge tdy = 1) \\ 0, & if(\forall r \in ri_1^n : r = 0 \wedge r_{st} = wait) \\ rdy, & otherwise \end{cases}$$

tdy' is defined by:

$$tdy' = \begin{cases} 0, & if(t_{st} = idle \wedge rxrdy = 1) \\ 1, & if(\forall a \in ai_1^n : a = 1 \wedge t_{st} = wait) \\ tdy, & otherwise \end{cases}$$

Handshake-based Architecture - Correctness Preserving Refinement

The functional definitions presented for the processes in the handshake-based architecture are deterministic by construction. Instead of repeating the proof in [**?**], we refer to the deterministic input/output (I/O) automata [155], where the composition of such deterministic (I/O) automata has been shown to preserve the Kahn properties [156]. An I/O automata is a formalism used to describe and reason about networks of concurrently executing processes. An I/O automata consists of input actions, output actions and internal actions, with the requirement that all input actions are always enabled. This means that whenever inputs are realized, the automata can appropriately react to them.

Theorem 3. *[156] If M is a deterministic I/O automata, then it is continuous and monotonic.*

We therefore transform our processes to equivalent I/O automata [155].

Consider $\mathcal{P}^h$ to be the set of all processes that are encapsulated using function hba in Section 8.4.1. Recall that for a given process $p' \in \mathcal{P}^h$ with n input data signals, we add n input request signals and n output acknowledge signals. Also, for m output data signals, we add m output request signals and m input acknowledge signals.

E_{hba} is the set of events in the handshake-based architecture, where input events ($E_{in} \subseteq E_{hba}$) are events on input data, request and acknowledge signals, and output events ($E_{out} \subseteq E_{hba}$) are events produced on output data, request and acknowledge signals. The local events ($E_{loc} \subseteq E_{hand}$) are the events generated within the process by receiver and transmitter units.

Theorem 4. *All processes in the handshake-based architecture are monotonic and continuous.*

Proof. (Proof by construction) The following procedure translates our encapsulations to I/O automata.

Step 1: All input events E_{in} are transformed to input actions.
Step 2: All output events E_{out} are transformed to output actions.
Step 3: All local events E_{loc} are transformed to internal actions.
Step 4: Define $Q \subseteq S_d \times S_{req} \times S_{ack} \times ST_{rx} \times ST_{rx} \times ST_{tx} \times rdy \times tdy$, where $q_o \in Q$ is the initial state with $q_o = (0, 0, 0, idle, idle, 1, 0)$. Notice that a state consist of various sub-states where, $data = 0$,$req = 0$,$ack = 0$, $rx_{st} = idle$, $tx_{st} = idle$, $rdy = 0$, and $tdy = 1$.
Step 5: Q consists of all the states the system such that T, the transition relation, is defined as $T \subseteq Q \times E_{in} \times Q$. $\forall e \in E_{in}$, $\exists\, q_i, q_j \in Q : q_i \times e \times q_j \in T$. This is true because for each sub-state, the values are computed based on the function definitions given. The function definitions described ensure that all inputs are acceptable because all definitions have the $otherwise$ condition, which holds when all previous defined conditions fail. Hence the values for any state can be computed as q = $(data = dout(..), req = txreq(..), ...)$.

Hence, we can create deterministic I/O automata for our processes, and by Theorem 3, we can conclude that our processes are continuous and monotonic. □

Next, we show that our refinement is correct-by-construction. To ensure this, we need to identify the events of interest in this architecture. By the event of interest, we mean that a new data value is realized on the data signal. An event of interest is identified whenever the request signal changes from 0 to 1. This means that a new request is made by the source process. Hence, we can assume that for every signal s, we can create a new signal $\bar{s}$ which is a sequence of events of interest.

Theorem 5. *Consider $p \in \mathcal{P}$ and its encapsulation $p' \in \mathcal{P}^h$, where di_1^n are input signals of p and $di_1'^n$ are input signals of p'. $\forall d_k \in di_1^n$ and $d'_k \in \bar{di}_1'^n$, if $d_k \equiv_e d'_k$, then $do_k \equiv_e do'_k$, where do and do' are their corresponding output signals.*

Proof sketch: We know that each process $p \in \mathcal{P}$ has a blocking read policy, whereas the write policy is non-blocking. However, each $p' \in \mathcal{P}^h$ has a blocking read and blocking write. The receiver in each process imposes a blocking read. This means that for each d'_k, if the events are of interest, then the data is passed to the computation block. The same applies for process p, and hence the output signals are latency equivalent. Now, when the writing blocks, the transmitter stalls the receiver from producing any new values. This ensures that the data produced is not overwritten by new data.

Theorem 6. *Handshake-based architecture is correct-by-construction.*

Proof sketch: We know that for a one process, if inputs are latency equivalent to its corresponding kahn process, the corresponding outputs are also latency equivalent (By theorem 5). Now, when we compose this with another process, we know that its output is latency equivalent to its corresponding kahn process. Hence, by induction, we can show that data is not lost between processes. Therefore, the handshake-based architecture is correct-by-construction.

A request signal being high ($req = 1$) implies that valid data is available on its corresponding data signal, and acknowledge signal being low ($ack = 0$) implies that its corresponding destination process is ready to accept data on its input. Now when both these conditions are true for all input and output signals for a process, then the communication occurs. The blocking write ensure that the process will only accept inputs when it is ready. This means that the state machines of the transmitter and receiver units of the process are in idle state and all its input signals have request high. So when the processes are composed in a network, it is guaranteed that data would not be lost between the processes. Therefore, we can conclude that our procedure is correct-by-construction. However, in the case of inputs and outputs, a fairness assumption on the request/acknowledge signals for inputs/outputs of the network is assumed.

Pros and Cons: For the handshake-based GALS architecture, we use the earlier proposed signaling protocol for communication between the components. This has been used for static dataflow architectures [35]. In both cases, at most one valid data value can be present on a communication signals. This is one of the cons of the architecture since the components would only execute if new data can be stored on the outputs. This also restricts parallelism in the design. Multiple handshakes are required for transferring data from one component to another which consumes more power and limits the performance. Secondly, if there are n inter-component signals, then $2 * n$ additional signals are required for request and acknowledgements. For the example shown, a total of 12 (4+2*(4)) signals are required.

8.5 FIFO-based GALS Architecture

We discuss two variants for implementing a FIFO-based GALS architecture. These are based on (i) handshaking scheme, and (ii) principles of LIP.

In the handshaking scheme for FIFO-based GALS architecture, the components are refined with the protocol discussed in the previous section, where RTUs are added to all components. An asynchronous FIFO with a bounded size is placed between the components. The component now handshakes with this bounded FIFO. We explain this with an example shown in Figure 8.5 where two components A and B communicate with an N-size FIFO in between.

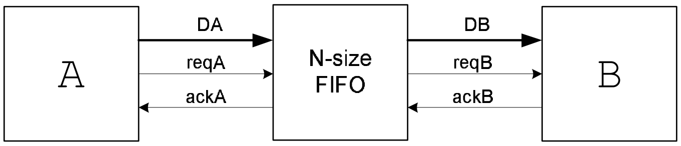


Figure 8.5: Asynchronous FIFO with handshake

The asynchronous FIFO placed in between two components (A and B) will require a RTUs on both its ends. Component A's RTU will communicate with RTU of the FIFO facing towards A. The RTU of the FIFO facing B will communicate with the RTU of B. For the four-phase handshake protocol, four handshakes will be required to communicate a single data from component A to FIFO, and the same from the FIFO to component B. In other words, a total of 8 handshakes will be needed to communicate a data from component A to component B. In the case of two-phase handshake, the total handshakes for exchanging one data will 4. Such an architecture will be very expensive with respect to the performance of the design.

We now propose a new GALS architecture based on the principles of LIPs [68, 70]. Recall that the communication is handled by valid-stall signals which are generated on the clocks of the components. Valid and stall signals are added for each inter-component signal.

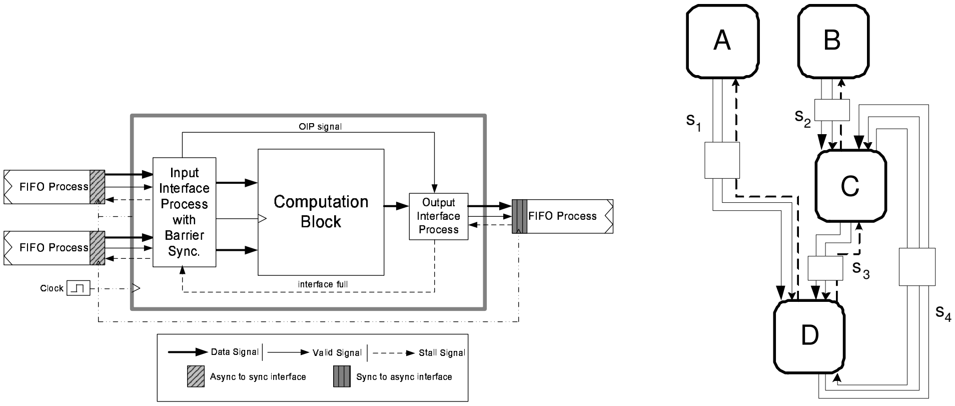
The protocol involves refinement of each component with: (1) Input interface process with barrier synchronization (IIP), (2) output interface process (OIP). Asynchronous FIFOs are placed between two components for communication. These FIFOs are equipped with interfaces that ensure correct communication between components with independent clocks. Detail information about these FIFOs can be found in [79]. Figure 8.6(a) illustrates the block diagram of a component in this architecture.

Input interface process with barrier synchronization (IIP): The IIP is placed at the input of the synchronous component. The main idea of this process is to barrier synchronize (align) all the valid inputs for the computational block. The block can only execute once all the inputs have been realized. Each IIP contains buffers for each input signal to store input data values. There are exactly two storage elements for each input. This is because when the computation block is stopped by the IIP, the incoming inputs need to be stored, and the stall signals for the appropriate source components have to be enabled. The need for the stall signal is realized as soon as the first storage element is filled. By the time the stall signal is enabled, the source component could have placed another valid value on the signal. Therefore, the second storage is needed to store this value. For each input to the component, IIP has an input data signal, an input valid signal and an output stall signal. Furthermore, it has two signals to communicate with OIP: (1) OIP signal which tells the OIP that a valid value has been processed by the computation block, and (2) Interface full signal from the OIP which tells the IIP that the OIP is ready to read a value or not. The IIP works in two phases: In the first phase it reads all inputs and stores the data values in its buffer. In the second phase it provides the data values to the component based on its input valid signals which are written to its output.

There are three possible scenarios that can occur:

1. All input valid signals are 1: This mean that valid values are seen on all inputs. During the first phase, the IIP stores the data, and at the start of the second phase, the values are provided to the computation block. Also, the IIP outputs a 1 on its OIP signal. The output stall signals to its source components are disabled (i.e. 0 is placed on them). The clock for the computation block is enabled.

2. One of the input valid signals are 0: This mean one of the input does not have valid data.



(a) Block Diagram of Component (b) FIFO based GALS

Figure 8.6: FIFO-based GALS Architecture

In the first phase, the IIP reads all the valid values from its inputs, and stores them in their respective storage elements. During the second phase, the IIP sends 0 on its OIP signal. This causes OIP to realize that the computation block has not produced a valid data. For the inputs where valid value was not realized, the IIP places 0 on their corresponding stall signals. All other output stall signals are enabled (set as 1). Now, in the following cycle, if the IIP realizes the valid values on all the inputs where valid was not seen in the previous cycle, then the procedure in scenario 1 follows. Otherwise, the stall signals remain unchanged for the inputs where valid values are not seen.

3. Interface full signal is 1 (enabled): This means that the OIP is not able to accept a valid data from the computation block. In the first phase, the IIP will read and store the inputs. In the second phase, the computation block is stalled, and the IIP outputs a 0 on OIP. The stall signals are set based on scenario 2.

Output interface process (OIP): The OIP is placed at the output of the synchronous component, and contains one buffer to store the result of the synchronous block. The inputs of OIP include valid value from the computation block, OIP signal from the IIP, and a stall signal from its destination FIFO. OIP reads and stores the value from the computation block whenever a 1 is received from the IIP (second phase). The OIP places the valid value received from the computation block to its output when the stall signal from the FIFO is disabled. In the case when the stall signal from the FIFO is enabled, the data from the computation block is stored in its buffer, and 1 is placed on the interface full signal to the IIP.

FIFO process: The $FIFO$ process provides the communication between two synchronous components. At each end of the FIFO, there are interfaces that communicate with the synchronous component. Note that this FIFO stands as an interface between the components running on different clocks. So, this FIFO has an synchronous to asynchronous interface on its input end, and an asynchronous to synchronous interface on its output end. Details about such interfaces can be found in [79]. The FIFO enables a stall to its source component when the buffer becomes full. Valid data is written on its output based on the clock of the destination component when valid value is present. Figure 8.6(b) illustrates a diagram of a FIFO-based GALS architecture.

Now, consider the example of FIFO-based GALS in Figure 8.6(b) and the clocks of its corresponding components in Table 8.2, we analyze its simulation trace which depends on the size of the FIFOs on the communication channels. Table 8.3 shows the size on the channels along with the number of valid values present on the channels. Note that the components execute on its clock when the data is present in the FIFOs of its input channels and its output channel FIFOs are not full.

-	FIFO size	t_0	t_1	t_2	t_3	t_4	t_5	t_6
s_1	2	0	1	2*	1	2*	1	2
s_2	3	0	1	0	1	1	2	3*
s_3	3	0	0	1	0	1	0	0
s_4	2	1	1	0	1	0	1	1

* denotes stall signal is enabled

Table 8.3: Count of valid values on channels.

The components $\mathcal{A}$ and $\mathcal{B}$ trigger on t_1, i.e. $\mathcal{A}$ and $\mathcal{B}$ will produce a valid value and store it in its output FIFO channels s_1 and s_2. At t_2, the clocks of components $\mathcal{A}$, $\mathcal{C}$, and $\mathcal{D}$ arrive. Component $\mathcal{C}$ executes as it has valid values on both its input signals s_2 and s_3 which are realized at t_1, and the token is removed from their respective FIFOs (recall that the input from $\mathcal{D}$ to $\mathcal{C}$ has an initial valid value as shown in t_0). Component $\mathcal{D}$ will not execute since no input is received from $\mathcal{C}$ at t_1. The count of valid value on s_3 is '0'. Component $\mathcal{A}$ produces another value which is stored in s_1. Now, the maximum FIFO size of s_1 is '2', so at this point, the channel reaches its maximum capacity. Therefore, a stall signal to component $\mathcal{A}$ is enabled to stop it from producing newer values (denoted by a *). The stall signal is disabled when the FIFO on s_1 is ready to accept more values. Note that the stall signals to the components are enabled/disabled by the FIFO.

8.5.1 Formal Definitions for FIFO-based Architecture

Let S_d denote the set of all data signals, S_v denote the set of all valid signals, and S_{st} denote the set of all stall signals for a system such that $S_d \cup S_v \cup S_st \subseteq S$. Each data signal is associated with a valid and a stall signal. A valid signal $v = v_1 v_2 v_3 ...$, where $v_i \in \{0,1\} \times \mathbb{N} \subset E$. If $v \in S_v$ and its corresponding $d \in S_d$, then $v[i] = 1$ iff $d[i]$ contains a valid data. We also assume that C is the set of all the clocks of the system.

The FIFO interface (between the processes) in this architecture contains a synchronous to asynchronous interface (write interface) on its input and asynchronous to synchronous interface (read interface) on its output. We denote the FIFO interface by $\mathcal{F}_{if}$. $\mathcal{F}_{if}$ is placed between two processes running on independent clocks. Hence, the interfaces of an $\mathcal{F}_{if}$ execute based on the clock of the process they connect to. This means that the clock of the process writing to the $\mathcal{F}_{if}$ and that of the write interface is the same, and also the clock of the process reading from the $\mathcal{F}_{if}$ and the read interface is same. As defined earlier, a FIFO is a list, however, it is bounded. The definition of the FIFO is similar to the one defined in Section 8.2. Each FIFO is bounded with a size that denotes the maximum number of values that can be stored. Once this FIFO becomes full, its read interface stalls its source component by its stall signal. Figure 8.7 shows the block diagram of the $\mathcal{F}_{if}$.

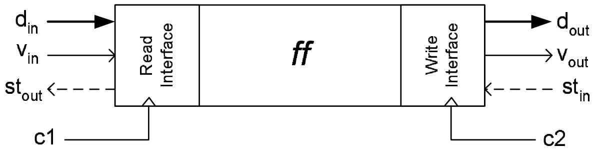


Figure 8.7: Block Diagram of Interface FIFO

Definition 36. *$iFF : S_d \times S_v \times S_{st} \times C^2 \times \mathbb{N} \rightarrow S_d \times S_v \times S_{st}$. Consider $di \in S_d$, $vi \in S_v$, and $sti \in S_{st}$ are the input data, valid, and stall signals. Assume $c1, c2 \in C$ are the clocks of the processes writing/reading from the $\mathcal{F}_{if}$. These are also connected to the corresponding read and write interfaces of the $\mathcal{F}_{if}$. $m \in \mathbb{N}$ defines the maximum size of the FIFO. Let ff denote the FIFO storage. The $\mathcal{F}_{if}$ outputs $do \in S_d$, $vo \in S_v$, and $sto \in S_{st}$. The new values of do', vo', sto' and ff' are computed by the functions $dout, vout, stout$, and $updtff$. Also, we assume that the parameters in these functions are current values of the corresponding signals. These values can be appended to respective signals.*

$$do' = dout(c_2, ff, do)$$

$$vo' = vout(c_2, ff, sti, vo)$$

$$sto' = stout(c_1, di, vi, ff, m, sto)$$

$$ff' = updtff(di, vi, ff, c_1, c_2)$$

where, the definitions of $dout, vout, stout$ and $updtff$ are given below.

$$dout(clk, ff, do) = \begin{cases} rdF(ff), & if(clk = 1 \wedge len(ff)! = 0) \\ do, & otherwise \end{cases}$$

$$vout(clk, ff, sti, vo) = \begin{cases} 1, & if(clk = 1 \wedge sti = 0 \wedge len(ff)! = 0) \\ 0, & if(clk = 1 \wedge (sti = 1 || len(ff) = 0)) \\ vo, & otherwise \end{cases}$$

$$stout(clk, di, vi, ff, sz, sto) = \begin{cases} 0, & if(clk = 1 \wedge len(addpr(di, vi, ff)) < sz) \\ 1, & if(clk = 1 \wedge len(addpr(di, vi, ff)) >= sz) \\ sto, & otherwise \end{cases}$$

$$updtff(di, vi, ff, clk1, clk2) = \begin{cases} addpr(di, vi, rmc(ff, clk2)), & if(clk1 = 1) \\ rmc(ff, clk2), & otherwise \end{cases}$$

$$addpr(di, vi, ff) = \begin{cases} enqueue(di, ff), & if(vi = 1) \\ ff, & otherwise \end{cases}$$

$$rmc(ff, clk) = \begin{cases} dequeue(ff), & if(clk = 1) \\ ff, & otherwise \end{cases}$$

Next, we consider the definition of the process encapsulation. The process encapsulation involves addition of an input interface (IIF) and an output interface (OIF). The inputs of the IIF are the data and valid signals from the FIFOs at the input, and an internal stop signal ($iifst$) from the OIF, whereas, the OIF takes a valid signal ($ivout$) from the IIF, data signals from the computation block, and stall signals from the FIFOs at the output. Note that $iifst$ is the interface full signal and $ivout$ is the OIP signal discussed in Figure 8.6(a). The IIF consists of a 2-buffer fifo for each input signal because by the time the stall signal is generated by the IIF, its source FIFO may have dispatched another data. On the other hand, the OIF consist of a 1-buffer fifo to store data in the case of stall from the destination FIFO. Figure 8.8 illustrates the block diagram of the process encapsulation.

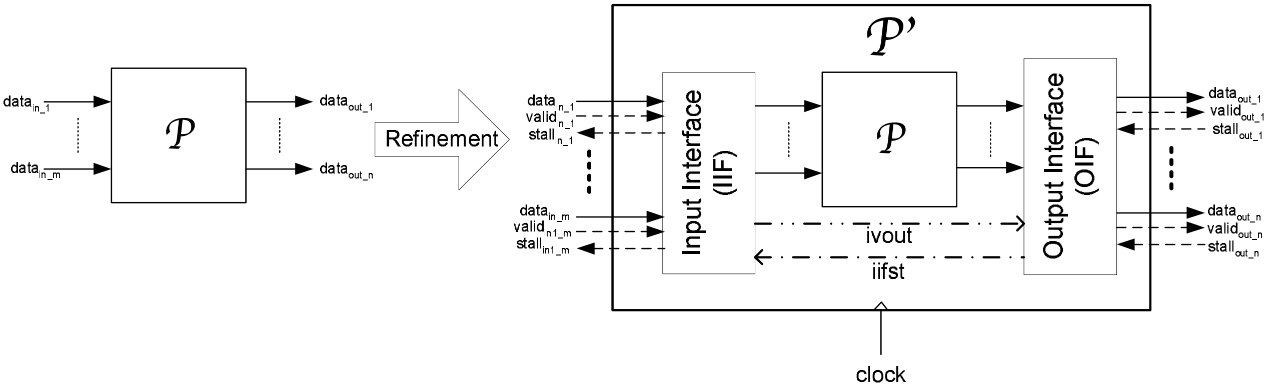


Figure 8.8: Block Diagram of the Process Encapsulation

The process in the FIFO-based architecture is defined as follows:

Definition 37. *$fba : S_d^n \times S_v^n \times S_{st}^m \rightarrow S_d^m \times S_v^m \times S_{st}^n$. Consider di_1^n, vi_1^n, and sti_1^m denote the input data, valid and stall signals, and do_1^m, vo_1^m, and sto_1^n denote the output data, valid and stall signals. Let $p \in \mathcal{P}$ denote the computation function being encapsulated. The encapsulation consists of IIF and OIF. The inputs to the IIF are di_1^n, vi_1^n and $iifst$, where di_1^n and vi_1^n are the data and valid inputs, and $iifst$ is the internal stop signal from OIF. The IIF consists of 2-buffer FIFOs denoted as fi_1^n for n input signals. The outputs of the IIF are dl_1^n, $ivout$, and sto_1^n, denoting the data signals to computation block, valid signal to OIF, and stall signals. On the other hand, the OIF takes inputs $p(dl_1^n)$, $ivout$, and sti_1^m, and outputs do_1^m, vo_1^m, and $iifst$. The OIF contains 1-buffer FIFOs, denoted as fo_1^m, for m output signals.*

The intermediate signals for the shell: $dl_k, ivout, fi$, and $iifst$ are defined by:

$$dl'_k = readF(addF2(di_k, vi_k, fi_k), iifst, dl_k)$$

$$ivout' = \begin{cases} 1, & if\ \forall k:\ vldF(addF2(di_k, vi_k, fi_k), iifst) = 1 \\ 0, & otherwise \end{cases}$$

$$fi'_k = \begin{cases} rmvF(addF2(di_k, vi_k, fi_k), iifst), & if\ ivout = 1 \\ fi_k, & otherwise \end{cases}$$

$$iifst' = \begin{cases} 1, & if\ \exists k : stF2(addF1(p(dl_1^n), ivout, fo_k), sti_k) = 1 \\ 0, & otherwise \end{cases}$$

$$fo'_k = rmvF(addF1(p(dl_1^n), ivout, fo_k), sti_k)$$

The output signals of the encapsulation are defined as:

$$do'_k = readF(addF1(p(dl_1^n), ivout, fo_k), sti_k, do_k)$$
$$vo'_k = vldF(addF1(p(dl_1^n), ivout, fo_k), sti_k)$$
$$sto'_k = stF1(addF2(di_k, vi_k, fi_k))$$

The helper functions used in the above definitions are defined as follows:

$$readF(ff, sti, do) = \begin{cases} rdF(ff), & if(len(ff) > 0 \wedge sti = 0) \\ do, & otherwise \end{cases}$$

$$addF2(di, vi, ff) = \begin{cases} enqueue(di, vi), & if(len(ff) < 2 \wedge vi = 1) \\ ff, & otherwise \end{cases}$$

$$addF1(di, vi, ff) = \begin{cases} enqueue(di, vi), & if(vi = 1) \\ ff, & otherwise \end{cases}$$

$$vldF(ff, sti) = \begin{cases} 1, & if(len(ff) > 0 \wedge sti = 0) \\ 0, & otherwise \end{cases}$$

$$rmvF(ff, sti) = \begin{cases} dequeue(ff), & if(len(ff) > 0 \wedge sti = 0) \\ ff, & otherwise \end{cases}$$

$$stF1(ff) = \begin{cases} 1, & if(len(ff) = 2) \\ 0, & otherwise \end{cases}$$

$$stF2(ff, sti) = \begin{cases} 1, & if(len(ff) > 0 \wedge sti = 1) \\ 0, & otherwise \end{cases}$$

8.5.2 FIFO-based Architecture: Correctness Preserving Refinement

In this section, we present the correctness proofs for the FIFO-based architecture. However, for all other architectures, similar techniques can be used to show their correctness.

We need to show that the refined architecture maintains the Kahn principle. To prove this, we show (i) each process in the architecture is continuous and monotonic, and (ii) the behaviors of the processes in the GALS architecture is correct with respect to its KPN counterpart. Our notion of correctness is latency equivalence.

We first present some basic definitions that we use to show correctness.

Definition 38. Sequential Composition (|): *Given two processes $p_1 : S^u \rightarrow S^v$ and $p_2 : S^v \rightarrow S^w$, and $s_1 \dots s_u \in S$, where $u, v, w \in \mathbb{N}$, we define the sequential composition as follows:*

$$p_1(s_1, \dots, s_u) \mid p_2 = p_2(p_1(s_1, \dots, s_u)).$$

The sequential composition with respect to our components definition can be extended such that the data signals, valid signals and stall signals are connected appropriately.

The function $strip$ given a signal with n events, returns a new signal without absent events.

Definition 39. ***Latency Equivalence****: The two signals s_1 and s_2 are said to be latency equivalent, $s_1 \equiv_e s_2 \Leftrightarrow strip(s_1, n) = strip(s_2, n)$, where $strip : S \rightarrow S$ be defined as, $strip(s, n) = \sigma(s, 1, n)$ and,*

$$\sigma(s,i,n) = \begin{cases} s[i] \oplus \sigma(s, i+1, n), & if\ (s[i] \neq \tau \wedge i \neq n) \\ \sigma(s, i+1, n), & if\ (s[i] = \tau \wedge i \neq n) \\ s[i], & if\ (s[i] \neq \tau \wedge i = n) \\ \text{[ ]}, & otherwise \end{cases}$$

The function σ takes the signal s, the start index i, and signal size n with respect to the number of events in the signal. The function outputs only valid events, and the absent events are removed. These definitions can be extended to processes also. For two latency equivalent processes, if their corresponding input signals are latency equivalent, then their output signals are also latency equivalent.

This definitions can be extended to processes also. For two latency equivalent processes, if their corresponding input signals are latency equivalent, then their output signals are also latency equivalent.

We first show each process in our FIFO-based architecture is continuous and monotonic. We again use the proof for I/O automata as discussed in Section 8.4.1. We show that the processes of our architecture can be correctly transformed into deterministic I/O automata.

Consider $\mathcal{P}^f$ to be the set of all processes that are encapsulated using definition 37 in Section 8.5.1. Recall that each process $p' \in \mathcal{P}^f$, n input data signals are associated with n input valid signals and n output stall signals. Also, m output data signals are associated with m output valid signals and m input stall signals. To map a signal to a sequence of actions, we consider the sequence of events on the signal.

Let E_{fifo} is the set of events in the fifo-based architecture. The input events are denoted by (E_{in}), output events are denoted by (E_{out}) and local events are denoted by (E_{loc}). Here, the input events are events seen on the input data, valid and stall signals, and the output events are events seen on the output data, valid and stall signals. The local events are events with in the process, between the IIF and OIF. Also, the definitions provided for encapsulations are deterministic by construction.

Theorem 7. *All processes in the fifo-based architecture are continuous and monotonic.*

Proof sketch: The following procedure transforms our encapsulations to I/O automata.

Step 1: All $e \in E_{in}$ are transformed to input actions.
Step 2: All $e \in E_{out}$ are transformed to output actions.
Step 3: All $e \in E_{loc}$ are transformed to internal actions.
Step 4: Define $Q \subseteq S_d \times S_v \times S_{st} \times \mathcal{F} \times \mathcal{F} \times dl \times ivout \times iffst$.
Step 5: Define $q_o \in Q$ as the initial state where, $q_o = (0, 0, 0, [], [], 0, 0, 0)$.
Step 6: Q consists of all the states the system such that T, transition relation, is defined as $T \subseteq Q \times E_{in} \times Q$. $\forall e \in E_{in}$, $\exists\ q_i, q_j \in Q : q_i \times e \times q_j \in T$. Hence, $q_i \times e \rightarrow q_j$. For $q_i = (d, v, st, fi, fo, dl, ivout, iffst)$ and $e = (di, vi, sti)$, $q_j = (d', v', st', fi', fo', dl', ivout', iffst')$ is computed as

$$d' = readF(addF1(p(dl), ivout, fo), sti, d)$$
$$v' = vldF(addF1(p(dl), ivout, fo), sti)$$
$$st' = stF1(addF2(di, vi, fi))$$

$$fi' = \begin{cases} rmvF(addF2(di, vi, fi), iifst), & if\ ivout = 1 \\ fi, & otherwise \end{cases}$$

$$fo' = rmvF(addF1(p(dl), ivout, fo), sti)$$
$$dl' = readF(addF2(di, vi, fi), iifst, dl)$$

$$ivout' = \begin{cases} 1, & if\ (vldF(addF2(di, vi, fi), iifst)) = 1 \\ 0, & otherwise \end{cases}$$

$$iffst' = \begin{cases} 1, & if\ (stF2(addF1(p(dl), ivout, fo), sti)) = 1 \\ 0, & otherwise \end{cases}$$

Since our definitions are deterministic, we can create deterministic I/O automata for our processes. Also, the definitions of the FIFO components can be extracted from the I/O automata. By equivalence of these components to I/O automata and theorem 3, we can conclude that our processes are continuous and monotonic. A similar proof for the interface FIFOs can be realized.

Similar to I/O automata, the refined components also have the receptive property that demands that each input action has a specified transition. When an input action occurs, the refined components does not need to wait on other inputs to arrive. Based on the definitions, data is either stored in buffers or a stall signal is enabled.

Next, we show that these refinements are correct-by-construction.

Theorem 8. *Consider $p \in \mathcal{P}$ and its corresponding $\bar{p} \in \mathcal{P}^f$, where di_1^n and $\bar{di}_1^n$ are their corresponding input signals, and do_1^n and $\bar{do}_1^n$ are their corresponding output signals. If for each k, $di_k \equiv_e \bar{di}_k$ then $do_k \equiv_e \bar{do}_k$.*

Proof sketch: We know that the data output of IIF in $\bar{p}$ is latency equivalent to $\bar{di}_1^n$ (By definition, IIF enforces blocking read and hence, barrier synchronizes its inputs). Since p has blocking read, the output of the computation block p and that of p' are the same. Next, we consider blocking write in our architecture. There are two scenarios that can occur: (i) The destination components are ready to accept new data: In this scenario, the OIF will produce $do_1^n = \bar{do}_1^n$, and hence $do_k \equiv_e \bar{do}_k$. (ii) The destination components are not ready to accept new data: In this case OIF will stall IIF, and hence no new data will be produced. As a result, the data will not be lost. Assuming fair environment, the destination components will eventually become ready and OIF will be able to provide data, and enable the IIF, thus producing new data[3]. Hence $do_1^n \equiv_e do_1'^n$.

Theorem 9. *For $p_1, p_2 \in \mathcal{P}$ and $p_1', p_2' \in \mathcal{P}'$, if $p_1 \equiv_e p_1' \wedge p_2 \equiv_e p_2' \Longrightarrow p_1 \mid p_2 \equiv_e p_1' \mid p_2'$*

Proof sketch: Since $p_1 \equiv_e p_1'$, for any latency equivalent input data signals, their output data signals are also latency equivalent (By theorem 8). Now, these latency equivalent signals are inputs to corresponding p_2 and p_2'. Hence, their outputs are also latency equivalent. By induction, this can be proved for any number of process compositions.

Pros and Cons: The number of valid and stall signals added would increase from the handshake based GALS architecture because of the FIFOs placed in-between the components. However, the components in this architecture may not necessarily stop after every execution. A component will only get stalled when no data is seen on any of its inputs, or if the FIFO buffers at its output channels become full. The stall signals form a back pressure mechanism that ensure that the data is not lost during communication [67]. This type of architecture is closely related to the static dataflow architecture with the difference that more number of tokens can be stored on the channels. The FIFO-based architecture will have a better performance with respect to the handshake-based architecture, and increases parallelism.

8.6 Controller-based GALS Architecture

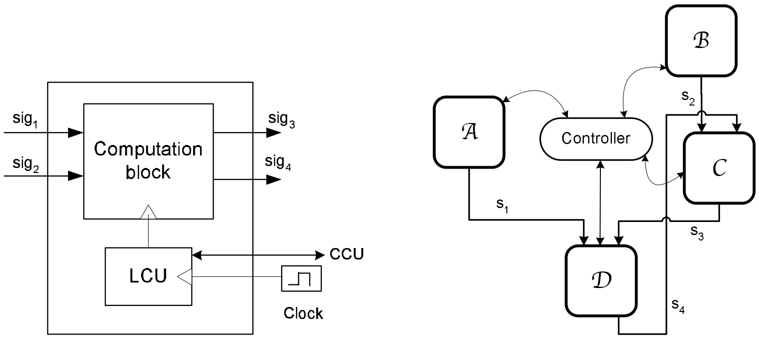
The controller-based GALS architecture is realized by refining each process in a KPN network into a synchronous component with a local control unit (LCU). The LCUs of the components communicate asynchronously with a central control unit (CCU) to request for a permission to execute. Figure 8.9(a) shows the block diagram of a component with an LCU unit, and Figure 8.9(b) illustrates a controller-based architecture.

The execution of the computation block is controlled by its LCU. The LCU sends a request message to the CCU. The format of the request message is as follows:

RequestMsg = { Component_id: String; Component_Status: boolean; Execution_Status: boolean;
Input_Signal_list: String list; Output_Signal_list: String list;}

The $Component\_id$ contains a unique name of the component. The $Component\_Status$ can be $true$ or $false$. A $Component\_Status = true$ means that the component is requesting for a

[3] Fair inputs to the components imply that their outputs are also fair (Can be realized from their definitions).



(a) Component with LCU (b) Controller-based GALS

Figure 8.9: Controller-based GALS Architecture

grant status, whereas a $Component\_Status = false$ means that the component is requesting for the update. The $Execution\_Status$ contains information about the previous grant request. This information is used by the CCU for updating its local structure[4]. The $Input\_Signal\_list$ and $Output\_Signal\_list$ contain the inputs and outputs of the component.

Algorithm 1 LCU execution steps on clock arrival

```
Step 1: Initialize the request message structure.
Step 2: Send request message to CCU with $Component\_Status = true$
Step 3: **If** grant=$true$
                Enable computation block for execution.
                $Execution\_Status = true$
        **else** $Execution\_Status = false$
Step 4: Send request message to CCU with $Component\_Status = false$.
```

The request signal passes the address of the $RequestMsg$ structure to the CCU with $Component\ \_Status = true$. The CCU upon receiving the address of the request message, retrieves the information and responds by giving a grant as $true$ or $false$. An enabled grant request has grant=$true$, otherwise vice versa. When the LCU receives a grant=$true$ from CCU, it enables the computation block for execution. After execution, $Execution\_Status$ is set to $true$ and $Component\_Status$ is set to $false$, and the request signal is sent back to the CCU. If grant=$false$ is received from CCU, $Execution\_Status$ as well as $Component\_Status$ are set to $false$, and the request signal is sent back to the CCU. The algorithm 1 defines the steps of the LCU that occur on each clock of the component. This is because the components in GALS only fire on the arrival of their clocks.

Next, we discuss the functionality of the CCU. The CCU is an asynchronous component that receives the request messages from the LCUs of different components, and based on the presence of values on the signals, grant the requests accordingly. The CCU consists of a simple structure that stores the presence and absence of values of different signals of the network. The storage

[4] We will discuss this later

structure for CCU is as follows:

SignalStatus = { Signal_Name: String; Value_Status: Boolean; }

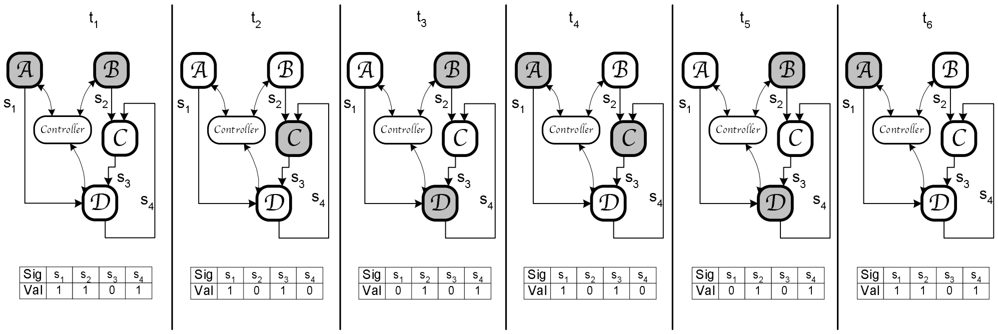
The $SignalStatus$ structure is stored as a list of structures. An alternate implementation can be organizing the same data as a hash table. The $Signal\_Name$ is associated with the signal connecting two synchronous components, and its $Value\_Status$ corresponds to a boolean value, which if high means that the signal has a valid value, and low means that the signal does not have a valid value. Algorithm 2 shows the steps taken by the CCU when it receives a request.

Algorithm 2 CCU execution steps on receiving request

```
If $Component\_Status = true$
        Fetch the appropriate status values from the $SignalStatus$ structure.
        If $SignalStatus$ values of all inputs are high, and all outputs are low
                grant=$true$
        else grant=$false$.
else if $Component\_Status = false$
        If $Execution\_Status = true$ (atomic step)
                Set all inputs to low in Signal_Status table.
                Set all outputs to high in Signal_Status table.
```

If more than one request is received by the CCU, the grant status is computed for all the requesting components. The update to the $SignalStatus$ structure only occurs if the message received from an LCU contains $Component\_Status = false$ and $Execution\_Status = true$. This update is done in an atomic step. Furthermore, when many requests are received by CCU, a case where two requests require updating the same signal value will never exists. This is due to the fact that the grant signals are always generated before the update is done, and the update occurs only on those signals that are either inputs or outputs to the components receiving true grant signals. So, if there are two components connecting each other, then they both will never be provided the grant request at the same time.

Now, consider the example of Controller-based GALS in Figure 8.9(b) and the clocks of its corresponding components in Table 8.2b, we analyze its simulation trace of the signal status table. Figure 8.10 shows the presence of values at each clock tick. The signals between the components and the controller handles the exchange of messages. Recall that we have initially assumed that signal s_4 has an initial value. The components that execute during the clock are shaded. For instance, at clock tick t_2, clocks of component $\mathcal{A}$, $\mathcal{C}$, and $\mathcal{D}$ arrive, however only $\mathcal{C}$ executes since s_1 and s_2 have valid values (realized at t_1). Components $\mathcal{A}$ and $\mathcal{B}$ do not receive an enabled grant signal from the CCU, as the values s_1 and s_2 are high in the signal status table at t_1. Secondly, if $\mathcal{A}$ and $\mathcal{B}$ were to be executed, then their previous values would have been overridden.



Sig	s_1	s_2	s_3	s_4
Val	1	1	0	1

Sig	s_1	s_2	s_3	s_4
Val	1	0	1	0

Sig	s_1	s_2	s_3	s_4
Val	0	1	0	1

Sig	s_1	s_2	s_3	s_4
Val	1	0	1	0

Sig	s_1	s_2	s_3	s_4
Val	0	1	0	1

Sig	s_1	s_2	s_3	s_4
Val	1	1	0	1

Figure 8.10: Simulation Trace of Signal Status Table in Controller

8.6.1 Formal Semantics of Controller-based Architecture

For this architecture, we use special signals that carry status messages from the processes to the CCU. We define such set of signals by $S_{smg} \subset S$. A signal $s \in S_{msg}$ denotes a sequence of events where the value part of the event contains a message consisting of the request status, execution status, input signals, and output signals. Hence, $e = ((rs, es, ils_1^n, ols_1^m), t)$ is an event, where rs is the request status, es is the execution status, ils_1^n is the list of the names of n input signals and ols_1^m is the list of names of m output signals of the process. rs and es are boolean values, whereas ils_1^n and ols_1^m are list of strings. The set of signals from the CCU to the processes is denoted by S_{gnt}, where $S_{gnt} \subset S$. A signal $s \in S_{gnt}$ carries the grant status in the form of $true$ or $false$, where the $true$ implies that the process can execute.

We define accessor functions for reads and writes from SST.

Definition 40. *The function $rdSST : SST \times S \to \{1, 0\}$, takes a signal s and returns its status value from SST.*

Definition 41. *The function $wrSST : SST \times S \times \{0, 1\} \to SST$, given a signal $s \in S$ and its new status st, updates the status of the signal s with st in SST.*

Given a set of signals, we define the function $ckirdy$ that returns if all the inputs have valid values.

Definition 42. *Consider the function $chirdy : S^n \to Bool$, where ils_1^n are n input data strings.*

$$chirdy(ils_1^n) = \begin{cases} true, & if\ \forall d \in ils_1^n : rdSST(SST, d) = 1 \\ false, & otherwise \end{cases}$$

Similarly, to identify that signals in SST contain invalid values, we define ols_1^m.

Definition 43. *Consider the function $chordy : S^n \to Bool$, where ols_1^m are m output data stings.*

$$chordy(ols_1^m) = \begin{cases} true, & if\ \forall d \in ols_1^m : rdSST(SST, d) = 0 \\ false, & otherwise \end{cases}$$

The CCU evaluates the events from different processes as it receives and returns appropriate grant signals. The behavior of the CCU is given below.

Definition 44. *The function $ccu : Bool \times Bool \times S^n \times S^m \rightarrow S_{gnt}$ takes in boolean values for rs and es, and names of input data signals ils_1^n and output data signals ols_1^m. The function returns the grant status for the process or updates its SST, based on the values of rs and es it receives. The function ccu is defined as follows:*

```
$ccu(rs, es, ils_1^n, ols_1^m)$ =
  if $(rs = true)$
          if ((for each $d \in ils_1^n : inprdy(SST, di) = true$)
          $\wedge$ for each $d \in ols_1^m : outrdy(SST, do) = true$)
              return $true$
          else return $false$
  else if $(rs = false)$
          if $(es = true)$
              foreach $di \in ils_1^n$ $writeSST(SST, 0, di)$
              foreach $do \in ols_1^m$ $writeSST(SST, 1, do)$
              return $false$
          else return $false$
```

Next we define the encapsulation of the processes. We assume that for each process $p \in \mathcal{P}$, its encapsulation $p' \in \mathcal{P}^c$ exists, where $\mathcal{P}^c$ is the set of all processes in the controller-based architecture. The encapsulation $\mathcal{R}^c : \mathcal{P} \rightarrow \mathcal{P}^c$ is defined by the function cba as follows:

Definition 45. *The function $cba : S^n \times S_{gnt} \rightarrow S^m \times S_{smg}$ takes in n input data signals di_1^n and one grant signal $g \in S_{gnt}$. The outputs of the process are m data signals do_1^m, and a status message signal $s_M \in S_{smg}$ to the CCU. We assume that $p \in \mathcal{P}$ is the computation function being encapsulated. The function lcu is defined as follows:* $cba(d_1^n, s_M)$ =

```
          if $ccu(true, false, ils_1^n, ols_1^m) = true$
              for each $do \in do_1^m, do' = f(di_1^n)$
              $ccu(false, true, ils_1^n, ols_1^m)$
          else for each $do \in do_1^m, do' = do$
              $ccu(false, false, ils_1^n, ols_1^m)$
```

Pros and Cons: In the controller-based GALS architecture, there is no back-pressure [67] mechanism which is seen in the FIFO-based GALS architecture and other existing GALS designs [140].

The synchronous components execute based on the grant requests received by the CCU. Also, each component has a simple communication model between the LCU and CCU for grant request. However, the CCU can be a major bottleneck for the design. This is because the request for all the components of the network are handled by this one single unit. Secondly, each component has back and forth (req/grant) signals to the CCU. The number of additional signals depend on the number of components in the design. The throughput of this architecture will be similar to the handshake-based architecture because at most only one token (valid value) can exist on a single arc (i.e. inter-component signal[5]). Furthermore, some of the ideas such as the use of a centralized controller have been borrowed from the tagged-token dataflow architectures [35].

8.7 Lookup-based GALS Architecture

In this architecture, we use an on-chip lookup storage unit (LUS) to store the data. This LUS is a bounded storage structure, and enables communication between different processes. Each process reads and writes to this LUS structure. Blocking read and blocking write conditions are enforced for each process by composing the process with a storage mapping unit (SMU). We discuss these in detail in the following sections.

8.7.1 On-chip Lookup Storage (LUS)

The data communicated between the processes is stored in the LUS. The LUS is split into different segments, where each segment represents a communication connection between two processes in the KPN model. The size of each segment depends on how many data elements can be stored before the consumer process starts processing the data. For n point-to-point connections in the KPN model, there are n segments in LUS unit. Each segment i has a bound sz_i associated with it. The segment size sz_i for each i connection denotes maximum number of elements that can be stored between two processes. Let SG represent the set of all segments of the storage, and $addr_k \in \mathbb{N}$ denote the address of segment k. Each segment k is associated with an index idx_k and a static bound bnd_k, where $idx_k, bnd_k \in \mathbb{N}$. The index points to the location of the data in the segment, and bound represents the maximum number of data elements that can be stored in the segment (or maximum value of the index).

Here, we do not focus on computing the optimal size for the storage, however, many existing works [**?**] focus on the buffer optimization problem between the processes. For data size $datasize$, the total size of the storage is computed as follows:

[5] A signal connecting two components

$$\sum_{i=1}^{n} sz_i * datasize = (sz_1 + sz_2 + \cdots + sz_n) * datasize$$

We now look at how the data is organized in the LUS unit. We assume that the most significant bit (MSB) of the data accessed represents the $present$ bit. The $present$ bit if set to 1 implies that the data is valid, otherwise it is invalid.

For example, consider data size to be 31 bits. So, 32 bit data is stored at each address location in the LUS, and its most significant bit (MSB) represents the $present$ bit. Figure 8.11 illustrates a segment of the LUS where between a producer and consumer at most x data values can be stored, where x represents some optimal size.

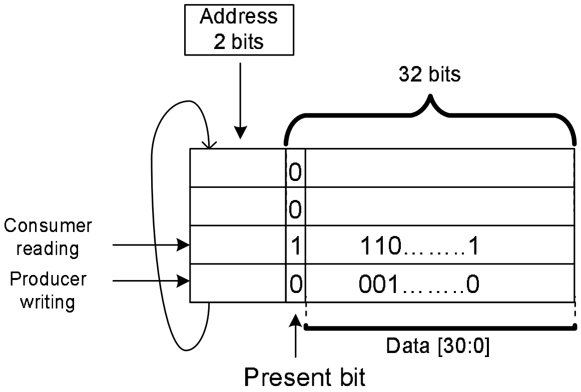


Figure 8.11: A Possible Segment Structure

LUS behavior: The read and write access of the LUS is a major constraint in the multiprocess model. LUS promotes asynchronous communication. A producer-consumer model as shown in Figure 8.12 is ideal to understand the LUS behavior. In an asynchronous environment, if a producer and consumer tries to access the same segment in a LUS, the access has to be granted in a manner which will protect the data integrity. The read and write operations of the LUS with respect to the producer-consumer model is discussed below. As LUS is divided into different segments for each point-to-point connection, each segment has a read controller and a write controller.

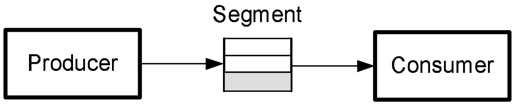


Figure 8.12: Producer-Consumer model for LUS

LUS Read Controller: The read access for the LUS is granted in a manner as shown in Figure 8.13. The read controller consists of four states: $Idle, PRead, CRead,$ and $PCRead$. The

$Idle$ read state of LUS denotes that there are no pending read requests. $PRead$ state denotes that producer's read request is being processed, and $CRead$ state denotes that consumer's read request is being processed. $PCRead$ state denotes that both the producer as well as the consumer are requesting read operation. A transition to these states denote that a request has been received. The read request from a producer and a consumer is denoted by $ProReq$ and $ConReq$ respectively. A transition out of these states denote that the read operation had been completed, and appropriate data has been sent back to the respective process. This operation completion to the producer and consumer of the segment is denoted by $/SendPro$ and $/SendCon$ respectively. For read operation, simultaneous read requests from producer and consumer can be handled together by granting both processes access to the data in the same segment.

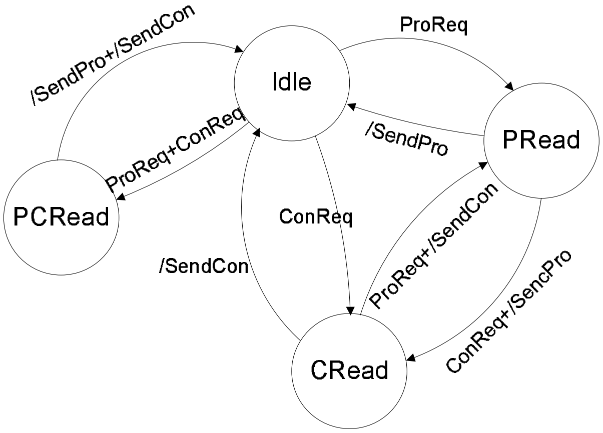


Figure 8.13: LUS Read Controller Behavior

LUS Write Controller: The write access for LUS has to be dealt with greater care than the read operation. We need to ensure that no read can happen to the same address in a segment during the write operation. Hence, the write requests are atomic. Figure 8.14 represents the behavior of how LUS handles the write requests. For write operation, LUS consists of three states: $Idle, PrCon$, and $WrCon$. The transitions $ProReq$ and $ConReq$ represents the write request from the producer

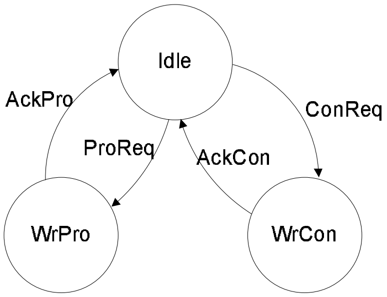


Figure 8.14: LUS Write Controller Behavior

and consumer. A write request is accompanied by the address and data, the the LUS writes the data in the corresponding address. $PrCon$ and $WrCon$ represents that the write operation is being performed for the producer and consumer respectively. One the write is complete, acknowledgement signals ($AckPro$ and $AckCon$) are sent to the processes to denote completion of write requests.

8.7.2 Process Refinement

The refinement involves composing each process $p \in \mathcal{P}$ of the KPN model with a storage mapping unit (SMU) that governs the execution of the process. The refinement yields $\mathcal{P}'$ where each $p' \in \mathcal{P}'$ is a refinement of its corresponding $p \in \mathcal{P}$. The refinement step is shown in Figure 8.15.

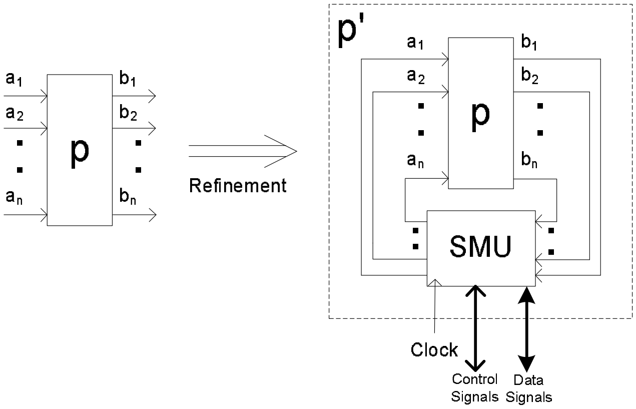


Figure 8.15: Process Refinement

The SMU consists of the addresses of the input and output data for its corresponding process. The SMU contains local storage elements to store the data for inputs/outputs of the process. This is based on the number of input and output fields. We assume initially that all storage is empty. The SMU also has the capability to extract the MSBs of the data retrieved. This can be implemented as a simple function. The SMU maps the addresses of each input/output to the correct lookup storage locations. Table 8.4 defines some of the basic operations for accessing the data.

Table 8.4: Lookup Operations

Function Name	Description
$rdloc(addr, idx)$	Function $rdloc$ returns the data at the address $addr$, and index idx.
$wrloc(addr, d, idx)$	Function $wrloc$ at the address $addr$ and index idx, writes the data d.
$rdprs(d)$	Function $rdprs$ reads the data d, and returns its present bit.
$wrprs(d, t)$	Function $wrprs$ writes the present bit t to the data d.

The SMU provides data to its corresponding process. For each input/output signal of the process, there is a segment in the LUS associated with the signal. Therefore, for each signal, there is a control and data signal connecting to the appropriate segment. Next, we define the storage structure of the SMU.

Data Structure: The storage structure contains fields for the inputs and outputs. Each input and output field is divided into two parts: address and bound. The address part points to the location of the inputs/outputs in the LUS. Initially, the address part for each input and output field contains its starting address. The bound part represents the maximum number of valid data locations that can be stored starting from the initial address location. In other words, the bound represents the maximum valid values that can be saved at a given time.

The storage structure in the SMU is shown in Figure 8.16. This organization helps in identifying the addresses of the inputs and outputs of the process simultaneously.

Address In1	Bound In1	Address In2	Bound In2	▬ ▬ ▬	Address Out1	Bound Out1	▬ ▬ ▬ ▬ ▬

Figure 8.16: Storage Structure in SMU

Functionality of the refined process: The behavior of the refined process p' is shown in Figure 8.17.

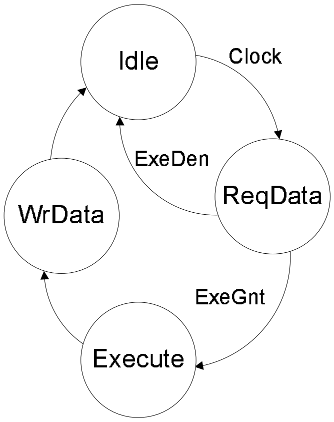
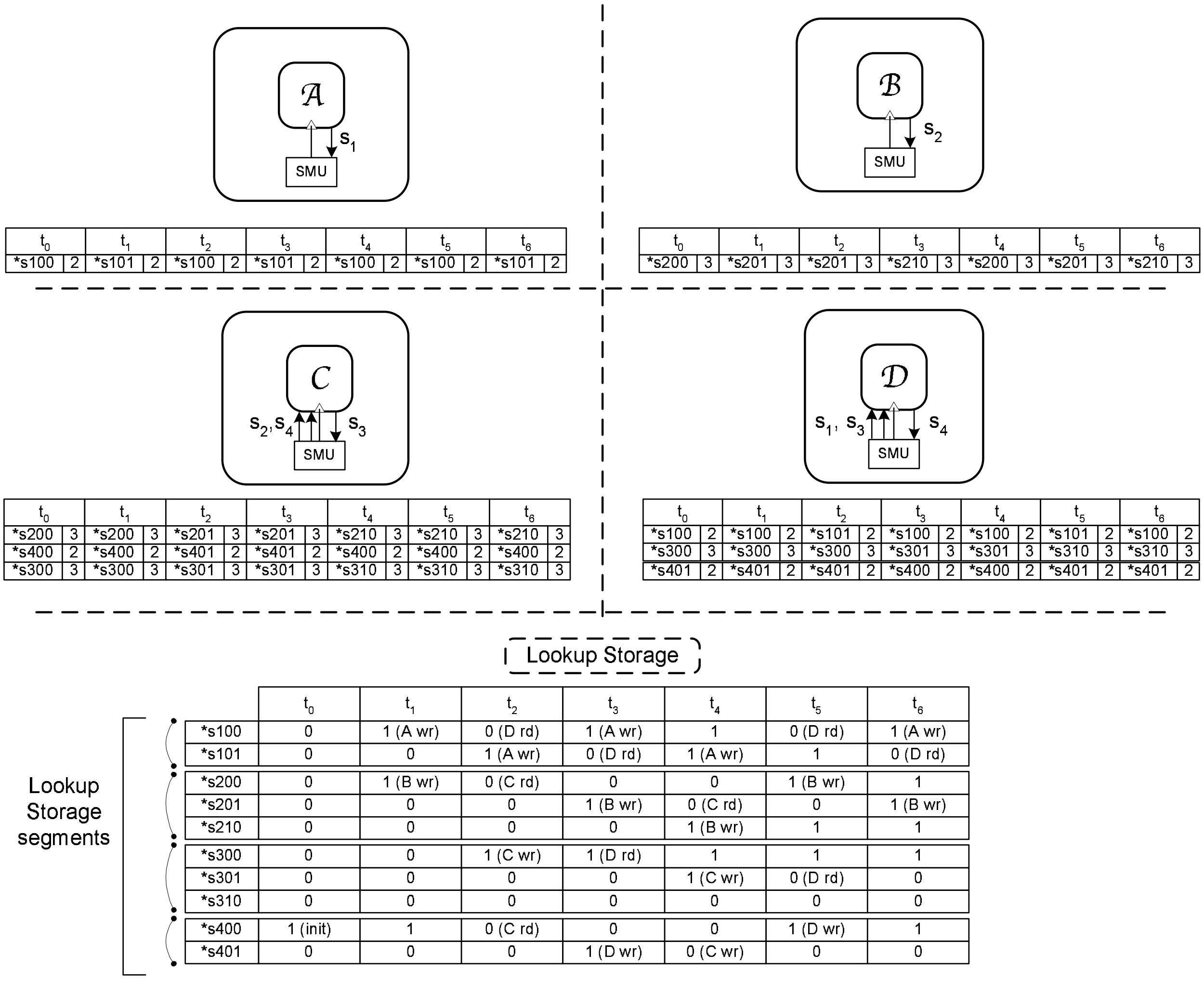


Figure 8.17: Process Behavior

The process can be in one of the four states: $Idle$, $ReqData$, $WrData$, and $Execute$. In the $Idle$ state, p' is waiting for the arrival of its clock. On arrival of the clock, the state of p' changes to $ReqData$. In this state, the following happens: (i) the SMU sends a request to LUS for retrieving data for the corresponding addresses of the input and outputs. (ii) Extract the $present$ bit of the data retrieved. (iii) If the condition that the $present$ bits of the data at all inputs are '1' and that of all the outputs are '0', then execution is granted ($ExeGnt$), and the transition to $Execute$ state is enabled. If the condition of the present bits is $false$ then the transition to the $Idle$ state is enabled. At the $Execution$ state, all the inputs are available and the output data can be produced. Hence, the data is sent to the process p for execution. After the execution, the output data is stored in the SMU. The transition to $WrData$ is enabled after the execution. In the $WrData$ state, the following happens: (i) The $present$ bits of the output data are set to '1'. (ii) The $present$ bits of the input data (data retrieved earlier) are set to '0'. (iii) The data is written back to the corresponding addresses of the input and output. (iv) The addresses of all inputs are incremented by '1' % $bound$ to point to the next read location, and the address of all outputs are also incremented by '1' % $bound$ to point to the next write location. (v) Once the acknowledgement is received from LUS, that data has been stored, the transition to the $Idle$ state is enabled.



t_0		t_1		t_2		t_3		t_4		t_5		t_6	
*s100	2	*s101	2	*s100	2	*s101	2	*s100	2	*s100	2	*s101	2

t_0		t_1		t_2		t_3		t_4		t_5		t_6	
*s200	3	*s201	3	*s201	3	*s210	3	*s200	3	*s201	3	*s210	3

t_0		t_1		t_2		t_3		t_4		t_5		t_6	
*s200	3	*s200	3	*s201	3	*s201	3	*s210	3	*s210	3	*s210	3
*s400	2	*s400	2	*s401	2	*s401	2	*s400	2	*s400	2	*s400	2
*s300	3	*s300	3	*s301	3	*s301	3	*s310	3	*s310	3	*s310	3

t_0		t_1		t_2		t_3		t_4		t_5		t_6	
*s100	2	*s100	2	*s101	2	*s100	2	*s100	2	*s101	2	*s100	2
*s300	3	*s300	3	*s300	3	*s301	3	*s301	3	*s310	3	*s310	3
*s401	2	*s401	2	*s401	2	*s400	2	*s400	2	*s401	2	*s401	2

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
*s100	0	1 (A wr)	0 (D rd)	1 (A wr)	1	0 (D rd)	1 (A wr)
*s101	0	0	1 (A wr)	0 (D rd)	1 (A wr)	1	0 (D rd)
*s200	0	1 (B wr)	0 (C rd)	0	0	1 (B wr)	1
*s201	0	0	0	1 (B wr)	0 (C rd)	0	1 (B wr)
*s210	0	0	0	0	1 (B wr)	1	1
*s300	0	0	1 (C wr)	1 (D rd)	1	1	1
*s301	0	0	0	0	1 (C wr)	0 (D rd)	0
*s310	0	0	0	0	0	0	0
*s400	1 (init)	1	0 (C rd)	0	0	1 (D wr)	1
*s401	0	0	0	1 (D wr)	0 (C wr)	0	0

Figure 8.18: Simulation of Lookup-based GALS Architecture

Now, consider the example of the lookup-based GALS (Figure 8.18) and the clocks of its corresponding components (Table 8.2b). For comparison purposes, we consider the bounds for each address to be the same as the corresponding size of the FIFOs considered in the FIFO-based architecture. The lookup-storage size can be computed by

$$\sum_{i=1}^{4} 32 * sz_i = 32 * (2 + 3 + 3 + 2) = 3200 bits = 400 bytes.$$

Figure 8.18 shows how the addresses are maintained in each component's SMU and how they change based on the arrival of clocks. As discussed earlier, the data is accessed (read for inputs and written for outputs) based on the local addresses. The SMU knows the appropriate segment where the addressees reside. In the example shown in Figure 8.18, we represent each address as *[signal name][location] for ease of readability. For instance, *s301 points to the appropriate location where the data of signal s3 is stored along with its offset 01. After every read/write by a component, the offset is incremented by 1 modulo bound of the signal. Also, for simplicity, the lookup storage structure shown only illustrates the presence (1) and absence (0) bits for the data. In actual storage, the data is read and written to these locations, along with the appropriate assignment of presence and absence bit to the MSB.

Pros and Cons: There are a few issues associated with this architecture. If a new clock arrives for a process when its state is not $Idle$, then clock does not have any affect on the state of the process. The process in such a case remains in the same state, and waits for the access. This does not change the behavior of the process, since the inputs are stored independent of the clock of the process. Second, as each signal in the KPN model is represented by a segment, only one process (producer or consumer) can write to the same segment of the LUS at any given time. The protocol at each process allows only one of these processes to write to the same address of the same segment. In the case when these processes write to different addresses of the same segment, then the write requests are handled sequentially by the LUS. In such a case the order of the write access does not make a difference, since the write is done to different addresses (of the same segment). Finally, meta-stability is avoided in this architecture as the read and write operations from LUS happen only after the arrival of the clock. If a new clock arrives during a read/write to the LUS, it is ignored. On the other hand, in a design with multi-clock FIFOs, meta-stability has to be addressed explicitly at the circuit level as the data can arrive during a clock transition. [79] addresses meta-stability issues in the multiclock FIFO interfaces for GALS.

8.7.3 Formal Semantics for Lookup-based Architecture

In this section, we present the correctness proofs for the lookup-based architecture. To ensure correctness, we show that the refined architecture maintains the Kahn principle by showing (i) each encapsulated process in our architecture is deterministic, continuous and monotonic, and (ii) the behaviors of the processes in the GALS architecture is correct with respect to its KPN counterpart. We again use *latency equivalence* as our notion of correctness (discussed in the earlier sections).

We first show each process in our architecture has the Kahn property, i.e. deterministic, continuous and monotonic. We again use the proof for I/O automata as discussed in Section 8.4.1. We show that the processes of our architecture can be correctly transformed into deterministic I/O automata. We first consider the refined process p'. Let $addri_1^n, addro_1^m$ denote the addresses of the n inputs and m outputs of the process, $idxi_1^n, idxo_1^m$ denote their corresponding indices, and bnd_1^n, bnd_1^m denote their corresponding bounds. We denote the data values for inputs and outputs as di_1^n and do_1^m. Assuming $do', di', idxi'$, and $idxo'$ denote newer values, and $p \in \mathcal{P}$ is the process being refined, the functionality of the refined process can be represented by the following functions.

The read operation for inputs and outputs is defined as follows, where LS represents the corresponding segment in the LUS unit.

$$\text{For each } addr \in addri_1^n,\ di_j = rdloc(LS_j, addr_j, idx_j)$$
$$\text{For each } addr \in addro_1^m,\ do_j = rdloc(LS_j, addr_j, idx_j)$$

Now, the following is the execution condition for the process:

$$\text{if } ((\forall di \in di_1^n\text{: } rdprs(di) = 1) \wedge$$
$$(\forall do \in do_1^n\text{: } rdprs(do) = 0))$$

If the condition is false, the refined process transitions back to $Idle$ state. If the condition holds true, then after process execution, the following behavior occurs.

For each k,

$$do'_k = wrprs(p(di_1^n), 1)$$
$$di'_k = wrprs(di_k, 0)$$
$$wrloc(addri_k, di'_k, idxi_k)$$
$$wrloc(addro_k, do'_k, idxo_k)$$
$$idxi'_k = (idxi_k\ + 1) mod\, bndi_k$$
$$idxo'_k = (idxo_k\ + 1) mod\, bndo_k$$

The definitions of the refined process presented are deterministic. The functions $rdloc$ and $wrloc$, read and write the data from the LUS unit, and are clearly deterministic. Furthermore, LUS contains data which can be modified by only $wrloc$ function, and hence is deterministic. Next, we show these units are continuous and monotonic.

Theorem 10. *The processes in our lookup-based architecture are continuous and monotonic.*

Proof sketch: We can easily transform the behavior of the refined process to its equivalent I/O automata. The inputs include arrival of events on the clock signal, arrival of data from read operation, and arrival of acknowledgement after write operations. These events are mapped to input actions. The output actions include the events on control signal and data signals to the LUS. The remaining events are mapped to internal actions. Now, to ensure that the inputs are enabled in all states, first we consider the arrival of the clock. If the clock arrives in the $Idle$ state, the transition is made to the $ReqData$ state. In all the other states, if the clock arrives, it remains in the same state. The input data actions are seen from LUS only in the $ReqData$ after an asynchronous event is sent to the LUS. In any of the other states, this event will not be seen as no read request is sent. Similarly, the acknowledgement (input action from LUS) will only happen after the write data signal is sent to the LUS. Hence, for all possible inputs, the I/O automata of the refined process takes appropriate actions, and hence can be represented as deterministic I/O automata. By $Theorem\ 3$, the processes are continuous and monotonic. Similarly, the LUS unit can also be represented by I/O automata, since all the actions that occur are asynchronous, and the appropriate deterministic responses are generated.

Next, we show that the behavior of our refined process $p' \in \mathcal{P}'$ is correct. With p', we also consider the corresponding segments where the data for its input and output is stored.

Theorem 11. *The behavior of a process $p \in \mathcal{P}$ and its refined process $p' \in \mathcal{P}'$ are latency equivalent.*

Proof sketch: The process p computes a valid value only when all its inputs are available (blocking read). The condition of the SMU ensure that the corresponding refined process computes when (a) all its inputs are available and (b) it is able to write data to its outputs. Now, if we only consider condition (a), then the behavior is equivalent to p. The condition (b) ensures that the data does not override an existing valid value. Once the p' consumer reads the existing valid value, it will make it invalid. So, for a single process, the consumer acts as an environment, and we assume that the environment is fair as in KPN model. Furthermore, since there is a fair environment assumption, the producer to p' will also not produce valid data until the data is consumed by p'. Hence, the output of p and p' will be identical for the same input.

Next, we show that the condition holds true for the composition of such processes.

Theorem 12. *The behavior of two composed processes $p_1, p_2 \in \mathcal{P}$ (p_1 and p_2 are connected by a point-to-point channel) is equivalent to the behavior of the composition of the corresponding refined process $p'_1, p'_2 \in \mathcal{P}'$ (p'_1 and p'_2 communicate via an asynchronous segment from our LUS unit).*

Proof sketch: Composing p_1 and p_2 together implies that one process is a producer and other must be a consumer. In the KPN model, if p_1 produces data on its output which is consumed by p_2. If X is a sequence of inputs, then $p_2(p_1(X))$ represents the behavior of this model. Now, consider their corresponding refined processes, p'_1 and p'_2, that communicate by a segment in the LUS. Using *Theorem 11*, we know that $p_1(X)$ and $p'_1(X)$ are equivalent. Now, since p'_1 and p'_2 share the same segment, p'_1 can write only at the address location where data has not been read by p'_2 (invalid data), and the address of p'_2 will point to the correct reading location (base assumption on how the initial address locations are assigned). Hence the outputs of $p'_2(p'_1(X)$ will be stored at locations where p'_2 will read from. As a result, the output of the GALS model will be $p'_2(p'_1(X))$. According to theorem 11, $p_2(p_1(X))$ and $p'_2(p'_1(X))$ are equivalent.

Using induction for the number of processes and its corresponding segments, we can prove that the behavior of the Kahn model and our GALS model is equivalent.

8.8 Comparison of the four architectures

Table 8.5 illustrates the execution of the components of our example for the four different architectures based on the clocks considered. Performance of the entire system can be analyzed based on how the components execute, and how many times the components execute. It can be realized that the handshake-based (using four-phase handshake) architecture had the worst performance, as each component executed twice in order to communicate one value across. Using a two-phase handshake for the handshake-based architecture would have improved this performance, but complicated the architecture. The drawback with the handshake based architecture is that there are twice more signals for each signal in the network. The performance of the controller based architecture was better than that of the handshake based architecture but worse than the FIFO-based and lookup-based architectures. Each component has signals going back and forth to the CCU, therefore for such an architecture, two signals are added to communicate with the CCU.

Next, we compare the FIFO-based architecture and the lookup-based architecture. In terms of performance, the lookup-based architecture is better, as each component does a fetch on its clock and stores the corresponding data in its local storage. As a result, the data in the storage has already been read. However, the main overhead for this approach is that there are many reads and writes to the storage for each component. In the case of FIFO-based architecture, the FIFOs are placed in-between the components. Similar to the handshake-based architecture, each signal is associated with two additional signals (valid and stall). However, the encapsulation of the computation block includes a barrier synchronizer, which functions the same as a join. Table 8.6 illustrates the

Architecture	t_1	t_2	t_3	t_4	t_5	t_6
Handshake (4-phase)	A,B	C	D	-	B	A
FIFO-based	A,B	A,C	B,D	A,B,C	B,D	A,B
Controller-based	A,B	C	B,D	A,C	B,D	A
Lookup-based	A,B	A,C	A,B,D	A,B,C	B,D	A,B

Table 8.5: Execution of Components in Different Architectures

overhead associated with the four architectures.

	Computation Complexity	Signal Overhead (n signals & m components)	Communication Media
Handshake-based	RTU	2*n	-
FIFO-based	IIP and OIP	4*n	FIFOs
Controller-based	LCU	2*m	CCU
Lookup-based	SMU	Control signals	Lookup Storage

Table 8.6: Overhead associated with GALS Architectures

The handshake-based GALS architecture should be chosen as the target architecture when there is a constraint on adding additional elements such as communication media (Table 8.6. Here, the cost associated with additional signals such as placement and routing is not an issue. FIFO-based architecture is a good choice as the target architecture for GALS, if additional signals can be added easily with FIFOs. Such an architecture would be best for performance driven applications. The controller-based GALS architecture is better if there is a constraint on number of signals can be added, and the ratio of the components in the design over the number of inter-component signals is higher. Hence, less number of signals will be added in this architecture than the handshake-based architecture. If addition of extra storage elements on the chip is not an issue, and storage accessing time is assumed to be little, then the lookup-based GALS architecture is best. It was realized by the example that the Lookup-based GALS architecture had the best performance if there are no constraints for additional elements/signals on the chip, and the accessing time was assumed to be negligible.

Chapter 9

Polychronous Methodology For System Design

In this spirit, synchronous programming languages implement a highly concurrent model of computation (MoC) in which time is abstracted by symbolic synchronization and scheduling relations to facilitate behavioral reasoning and functional verification. In the particular case of Polychrony [15], time is represented by partially ordered synchronization and scheduling relations, to provide an additional ability to model high-level abstractions of system paced by multiple clocks: globally asynchronous systems. Polychrony favors the progressive design of *"correct by construction"* systems by means of well-defined model transformations that preserve the intended semantics of early requirement specifications and eventually provide a functionally correct deployment on target architectures [122].

Unfortunately, the polychronous models are often very counterintuitive to regular engineers. Such models specify systems where variables are updated at independent rates resulting in a complex semantic theory based on 'clock calculus' [31, 32]. This complication in theory though led to powerful analysis capabilities, leading to optimized synthesis of embedded software systems which sample its environment exactly as required, no more, no less. Hence, if the semantic theory can be rendered more palatable to engineers, polychrony can be used by a much wider audience than computer scientists alone.

In order to simplify the notion of polychronous specification, we model a "true concurrency" representation of polychrony. *True concurrency semantics* are useful to analyze multiple behaviors that occur simultaneously, which in contrast is different from *interleaving semantics*, where such behaviors occur in an interleaved manner (cf. models in [130, 166, 86]). For example, consider two independent atomic actions α and β in a system which can occur concurrently. In a *true concurrency* model, the execution of α and β occur simultaneously. Now, in the interleaving model, the execution of α is followed by β or vice versa. Figure 9.1 shows the interleaving and true concurrency behavior of α and β. Since, polychrony is a concurrency model where activities happen at their own pace except when explicit synchronizations are required. We chose true concurrency as a semantic model to understand the semantics of polychrony and characterize various subclasses of polychronous programs. Polychronous programs can have many characterizations, such as deterministic programs [43], endochronous programs [43, 122, 41], endo-isochronous programs [43, 122], weakly-endochronous programs [186, 187], etc. As we will see in subsequent sections, some of these characterizations lead to easier implementation of the model as sequential programs, such as endochrony.

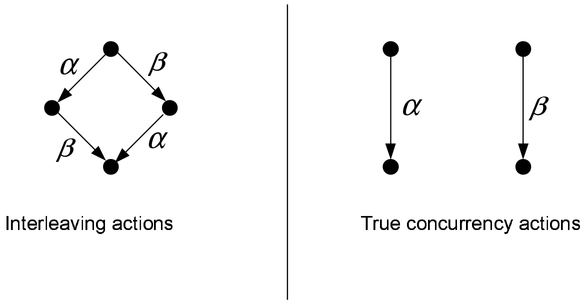


Figure 9.1: Interleaving and true concurrency behaviors

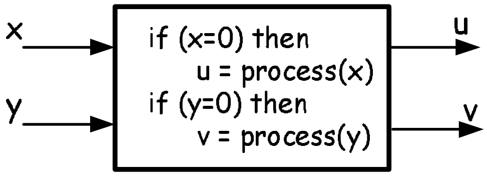
In this chapter, we show that this semantic approach not only provides intuitive characterization of determinacy, and endochrony [1] [43], but also helps in analyzing the synchronous and asynchronous interpretation of the same program in a single representation. We use the terms ‘synchrony’ to imply an implementation with instantaneous communication between processes, and ‘asynchrony’ to imply delayed communication based implementation. In this context, a synchronous interpretation means that an observer is observing the behavior with respect to successive reactions, whereas, an asynchronous interpretation means that an observer is observing the behavior as streams of input/output events.

The notion of ‘endochrony’ in polychronous model of computation is used to decide the feasibility of single clock implementation of a specification, or of a distributed implementation over asynchronous channels. However, due to the mathematical formalisms used to define such notions are highly abstract, and are difficult to comprehend intuitively for a given program.

Originally, polychrony was explained with ‘synchronous transition systems’ (STS) [43] but more recently polychronous programs have been given semantics in [122] using the *tagged signal model* of Lee and Sangiovanni [152]. This semantic model of polychrony has an artifact of tags with each new value of the variables/signals during the program execution. Our work here show that these tags are unnecessary for the purpose of characterizing polychronous programs. In fact their usage makes asynchronous interpretation of programs unduly complicated, resulting in difficult to understand theories for characterizing polychronous programs, such as characterization of endochronous programs [42, 122]. In this chapter, we bring forth the pomset based semantic model which is intuitive for engineers while it removes the need for a tagged-signal model.

[1] Defined in Section 9.2

In embedded software design, one big question is when can one place a program in an asynchronous environment and expect to obtain the correct output sequence, similar to what one would expect from the program in a synchronous environment. Such question is related to the issue of asynchronous interpretation of a program. Consider the example shown in Figure 9.2, where x, y are values generated by the environment, and v, u are values consumed by the environment. If x, y are generated by independent sources, more new values of x may come before a single new value of y or vice versa. If the program inside the box is sequentially written, then if x, y don't come at the same rate, while processing y, some new value of x may be missed. If the exact relative rate of arrival of x, y are known, one can write the appropriate multi-threaded code. Our semantic model is naturally an asynchronous interpretation, and its synchronous interpretation is constructible for deterministic polychronous programs by way of *"levelling functions"* defined in this chapter.



```
if (x=0) then
    u = process(x)
if (y=0) then
    v = process(y)
```

Figure 9.2: Program communicating with its environment

Interestingly, the same deterministic program may have multiple possible synchronous interpretations, and a choice one makes leads to a refinement of the polychronous specification. Multiple possible synchronous interpretations can occur when the environment generates independent inputs at multiple unrelated rates (*non-endochronous specifications*).

As a result, if sequential code is generated from such a polychronous specification, multiple possibilities will exist, and a choice of one over another will not be correct without having extra constraints on the environment. So if one is interested only in the class of programs that has correct and unique sequential implementation, one would be interested in the class of endochronous specifications.

Since generating "correct-by-construction" sequential embedded software from polychronous specifications is very important for engineers interested in model driven code generation, this chapter attempts to demystify the notion of endochrony in very simple terms. For example, endochronous systems have the property that they can be uniquely *desynchronized* and *resynchronized* [122], whereas one can also state that such systems can be easily classified as single clock triggered systems, which can have multiple rationally related clocks. Our pomset based representation of polychrony also attempts to simplify this understanding by providing a visual representation of the execution traces. Further characterizations of programs, especially for generating multi-threaded implementation will be a topic of further research.

Main Contributions: The main results of this chapter are as follows: (i) We show a true concurrency semantics for polychronous specifications using pomset [189], and furthermore construct semantic models from polychronous specifications written using the SIGNAL language [15]. In doing so, we interpret the polychronous specifications without the notion of tags. (ii) We formalize the property of endochrony in terms of pomset semantics, and show that they are intuitive and visually relatable in this formalism. We provide an illustrative example of pomset characterization of endochrony in terms of levellings.

9.1 Background

In this section, we introduce the background and preliminary definition necessary to understand our formalism. We define various terminologies used in the chapter, introduce the semantics of the tagged signal model for polychrony from [122], and briefly discuss the SIGNAL language and pomsets.

A *program* is a collection of statements that execute based on the operations defined by the statements. The statements can be of the form assignments, conditions, loops, etc.

Let $\mathcal{V}$ denote the universal set of all variables and D be the set of all data values.

9.1.1 Polychronous Model of Computation

Polychronous MoC [122] facilitates the description of systems in which components obey to multiple clock rates. The reason for such multiple clock rates is that such systems interact with the environment, and the rate at which environment generates values are not necessarily known unless constraints are imposed on the environment, which often is not possible. Polychrony also provides a mathematical foundation to a notion of refinement and the ability to model a system from the early stages of its requirement specifications to the late stages of its synthesis and deployment.

In the tagged signal based semantic model of polychrony [122], a partially-ordered set $(\mathcal{T}, \leq, 0)$ of tags is considered. A tag $t \in \mathcal{T}$ denotes a symbolic instant or a period in time. We denote $C \in \mathcal{C}$ a totally ordered subset of $\mathcal{T}$, where $\mathcal{C}$ is the set of all chains. Signals, behaviors and processes are defined starting as follows:

An *event* $e \in \mathcal{T} \times D$ is the pair of a tag and a value. The absence of an event, is denoted as ϵ. A *signal* $s \in C \rightarrow D$ is a function from a *chain* of tags to values. A *behavior* $b \in \mathcal{B}$ is a function from names $x \in \mathcal{V}$ to signals $s \in \mathcal{S}$. A *process* $\mathcal{P}$ is a set of behaviors that have the same domain.

We write $\mathrm{tags}(s)$ and $\mathrm{tags}(b)$ for the tags of a signal s and of a behavior b; $b|_X$ for the projection of a behavior b on $X \subset \mathcal{V}$ and $b/X = b|_{\mathrm{vars}(b) \setminus X}$ for its complementary; $\mathrm{vars}(b)$ and $\mathrm{vars}(r)$ for the domains of b and r.

The synchronous composition $r \,|\, s$ of two processes r and s is defined by the union of all behaviors b_r (from r) and b_s (from s) which carry the same values at the same tags.

$$r \,\|\, s = \{b_r \cup b_s \;:\; (b_r, b_s) \in r \times s, I = \text{vars}(r) \cap \text{vars}(s), b_r|_I = b_s|_I\}$$

Note that this definition implies that on the variables common to r and s, communication is instantaneous.

Example 4. *Consider two behaviors b_1 and b_2 with $vars(b_1) = \{a, b, c\}$ and $vars(b_2) = \{c, d, e\}$, and $\{a, b, c, d, e\} \in \mathcal{V}$. The synchronous composition of $b_1 \,\|\, b_2 = \{a, b, c, d, e\}$, where $b_1|c = b_2|c$.*

A *trace* consists of a sequence of events ordered based on their tags. A clock[2] for the *trace* can be derived based on these tags. Each variable is associated with a signal, hence represents a behavior. Therefore, each variable is associated with a clock that denotes the chronology of the occurrence of events. For a variable v, we denote its clock by $\hat{v}$. Consider the following example:

Example 5. *Consider the observation of the trace of variable a shown below.*

$tags:$	t_1	t_2	t_3	t_4	t_5
$a:$	5	11	3	1	10

The data values that are assigned to variable 'a' are $5, 11, 3, 1$, and 10, and their respective tags are t_1, t_2, t_3, t_4 and t_5. The clock $\hat{a}$ is $\langle t_1, t_2, t_3, t_4, t_5\rangle$, which is based on the occurrence of events for a.

A *p*olychronous system is a collection of *p*rograms associated with multiple clocks. The programs of such a system may or may not be related by rational clocks.

9.1.2 SIGNAL

A SIGNAL program [122] consists of several equations over signals, which are composed together. Each equation executes simultaneously in parallel. Assume that X is the set of all signal names of the program with signals $x, y, z, ...$, then an equation is defined as $x = f\ y$, where $f \in F$ is the process, and the equation defines the relation between the signals x and y. The SIGNAL language consists of three primitive processes: `pre`, `when` and `default`. The equation `x = pre val y` defines that the signal x holds the previous value of signal y, with the initial value set as val. The equation `x = y when z` defines that the signal x holds the value of signal y when signal z is present and true, otherwise the value of signal x is not defined. The equation `x = y default z` defines that the signal x holds the value of signal y when it is present, and otherwise holds the value of signal z. The semantics of the primitive processes are presented in [122]. The equations can also represent arithmetic operations. SIGNAL has the notion of *time instants*, where each time

[2] A clock denotes the frequency at which the events occur

instant denotes one reaction cycle. Signal also contains a delay operator $, that delays the events of a signal by n time instants.

For signals $x, y \in X$, we say x is *functionally dependent* on y if $x = f(y)$ for some operation f. The events of x are realized at the same time instants of y. Here, the clocks of all the signals are same. This type of functional dependency is implicit. However, considering operators that explicitly add delay between signals, we say these signals are *causal*. Causality induces temporal precedence between the events of signals. Consider the following example with signals $a, b, c \in X$. We define *b = a + 5* and *c = a$1*.

Example 6.

$tags:$	t_1	t_2	t_3	t_4	t_5	...
$a:$	5	11	3	1	10	...
$b:$	10	16	8	6	15	...
$c:$	$-$	5	11	3	1	...

In the example, the signal b is functionally dependent on signal a, and signal a causes the events of signal c with a delay of 1. The consequence of the application such operators induces a *scheduling* of these new events.

9.1.3 Pomsets

For pomsets, we follow the definitions of [189].

Definition 1. *A* pomset *[189] is an isomorphism class of a* labelled partial order (lpo) *defined as a 4-tuple as* $\langle V, \leq, \Sigma, \mu \rangle$ *where:*

- V *is the set of vertices modeling events. Each event ($\bar{e}$) in* V *is an instance of an action* $\in \Sigma$.
- $\leq$ *is the partial order defined on* V *which expresses precedence between events. If* $\bar{a}, \bar{b} \in V$, *then* $\bar{a} \leq \bar{b}$ *is interpreted as event* $\bar{a}$ *preceding event* $\bar{b}$ *in time. (So* $\leq$ *is really a pre-order).*
- Σ *is the alphabet modeling actions. For example an action can be an assignment of a data value to a variable (e.g reset:=true).*
- $\mu : V \rightarrow \Sigma$ *is a labelling function that assigns the actions to vertices.*

In this work, Σ and μ components play no direct role.

9.2 Pomset Representation of Polychrony

An event of a pomset is represented as $\bar{e}$, where $\bar{e} = (x, d)$, for $x \in \mathcal{V}$ and $d \in D$. An *event set* is a set of such events that are related by a partial ordering relation ($\leq$). A pomset is visualized as a graph, where each node corresponds an event from the vertex set V, and the arcs order the events based on ($\leq$). In the graphical representation, we only show arcs between two events $\bar{e}_i, \bar{e}_j \in V$, **iff** $\bar{e}_i \leq \bar{e}_j$, $\nexists \bar{e}_k : \bar{e}_i \neq \bar{e}_k \neq \bar{e}_j$ **s.t.** $\bar{e}_i \leq \bar{e}_k \wedge \bar{e}_k \leq \bar{e}_j$. We denote the partial order $\leq$ by $\mapsto$ if there is an edge between the event vertices in the pomset graph.

Example 7. *The pomset P for the trace of variable 'a' in example 6 is $\langle V_P, \leq_P, \Sigma_P, \mu_P \rangle$, where $V_P = \{\bar{a}_1, \bar{a}_2, \bar{a}_3, \bar{a}_4, \bar{a}_5\}$ and $\bar{a}_i = (a, i)$, $i = 1, 2, 3, 4, 5 \in \mathbb{N}$. The partial order, $\leq_P$ is defined on V_P, where $\bar{a}_1 \mapsto \bar{a}_2 \mapsto \bar{a}_3 \mapsto \bar{a}_4 \mapsto \bar{a}_5$, and Σ_P the action set is $\{a := 5, a := 11, a := 3, a := 1, a := 10\}$. μ_P is the labelling function that maps the events to their actions. For example, $\bar{a}_1$ is mapped to action $a := 1$. The pomset visualization of polychronous variable a is shown in Figure 9.3.*

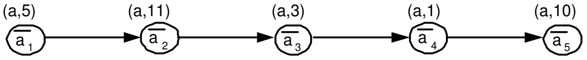


Figure 9.3: Visualization of pomset P

We now need to extend a pomset with the notion of *functional pomset*, that captures functional dependency between events in the set. The partial order only captures temporal precedence of events, and does not capture the dependency caused by the fact that some events occur due to a function computation on some other events. Later in the chapter, when we show how to construct a new pomset from pomset representations of two processes which interact in the sense that events in them participate together in a function computation, we need to introduce a new pre-order ($\leq^f$) on the resulting event set, that expresses the ordering based on functional dependency. If $\bar{b} = f(\bar{a})$ for $\bar{a}, \bar{b} \in V$, then $\bar{a} \leq^f \bar{b}$ is interpreted as event $\bar{a}$ precedes in functional sense event $\bar{b}$, and $\bar{b}$ is functionally dependent on $\bar{a}$.

Definition 2. ***Functional Dependence:*** *An event $\bar{a}_i$ is **functionally dependent** on an events $\bar{b}_{j_1}$, $\bar{b}_{j_2}$, ..., $\bar{b}_{j_n}$ if $\bar{a}_i$ = $f(\bar{b}_{j_1}, \bar{b}_{j_2}, \ldots, \bar{b}_{j_n})$. We denote functional dependence with $\leq^f$ and represent the corresponding edge in the pomset visualization with $\hookrightarrow$ arc.*

Example 8. *Consider the example for $z = f(x)$, where $f(x) = x + 5$. The traces of variables x and z are shown below:*

$time:$	t_1	t_2	t_3	t_4	t_5	t_6	t_7
$x:$	1	-	3	-	5	-	7
$z:$	6	-	8	-	10	-	13

In this example, every value of the trace of variable x is incremented by 5, and its result is shown in the trace of z. The new events are realized at the same time instants due to the synchrony assumption. *There is a functional dependence between the events of variables x and z and are written as $\bar{x}_i \hookrightarrow \bar{z}_i$.*

The addition of functional dependence order among events in a pomset means that we have a new semantic object which is a pomset together with another non-reflexive partial order $\leq^f$. For ease of writing, we call this functional pomset also pomset, with the understanding that a $\leq^f$ ordering may or may not be attached to it. If one considers, a single input variable of a program, and all the events happening for that variable, we usually get a totally ordered pomset or a tomset. Such a pomset does not have any $\leq^f$ associated with it. The functional dependence order ($\leq^f$) is considered when we construct the pomsets corresponding to a program that computes functions based on such input variables. In the following section, we show how we construct pomsets semantics for SIGNAL programs with causal as well as functional dependence. Throughout this chapter, for a given pomset p, we denote its partial ordering by $\leq_p$, and its functional ordering by $\leq^f_p$. *It should be clear that $\leq_p$ of a pomset p, and its $\leq^f_p$ are always disjoint.*

One important construction fact to know is how to capture synchronization between two events in the $\leq$. Let x and y be two variables in the program, and events happening in those variables are expressed as tomsets $\{x_1, x_2, ...x_i, ...\}$ and $\{y_1, y_2, ...y_j, ...\}$. Suppose the program requires that x_i and y_j to be synchronized. This can be captured by adding (x_i, y_{j+1}) and (y_j, x_{i+1}) in $\leq$.

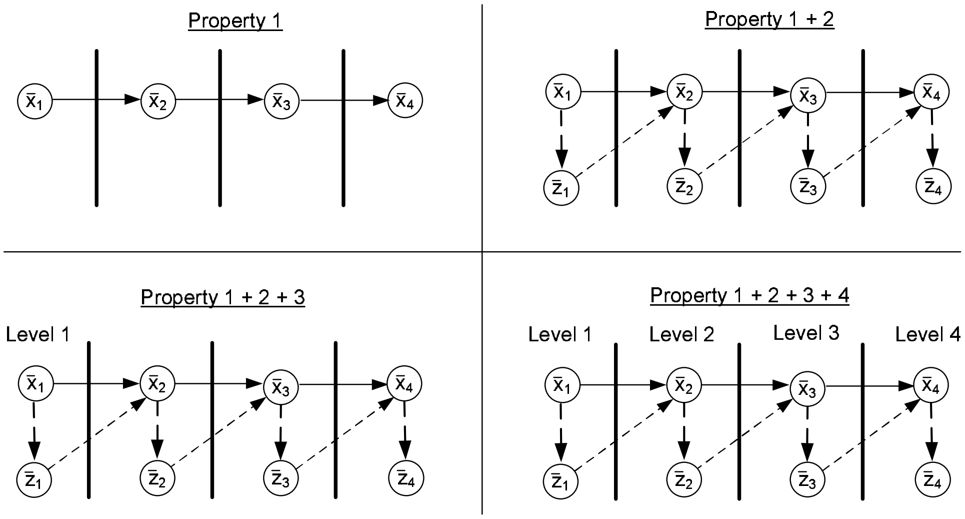


Figure 9.4: Levelization of traces of x and z

Furthermore, we define the notion of *levelization* of the events in the event set by using a level function l. *Levelization* involves arranging the events into levels based on their partial ordering.

Definition 3. ***Levelling Function:*** *Given a pomset $p : \langle V_p, \leq_p, \Sigma_p, \mu_p \rangle$ and its associated functional ordering $\leq^f_p$, a level function $l : V_p \rightarrow \mathbb{N}$ has the following properties:*

1. $\forall \bar{e}_i, \bar{e}_j$ **if** $\bar{e}_i \leq_p \bar{e}_j$, **then** $l(\bar{e}_j) > l(\bar{e}_i)$

2. Let $V_p(\bar{e}_j)$ = $\{\bar{e}_{i_1}, \bar{e}_{i_2}, ..., \bar{e}_{i_k}\}$ be the set of events **s.t.** $\forall \bar{e}_i \in V_p(\bar{e}_j)$

if $\bar{e}_i \leq_p^f \bar{e}_j$, $\nexists \bar{e}_k \notin V_p(\bar{e}_j)$ $(\bar{e}_k \leq_p^f \bar{e}_j)$, **then** $l(\bar{e}_j)$ = $max_{\bar{e}_i \in V_p(\bar{e}_j)}$ $l(\bar{e}_i)$

Intuitively, events that are ordered by $\leq^p$ should have increasing level numbers, and for a set of events that lead to a function computation, the level of the resulting event from the function computation will be max of all the levels in the contributing set.

Let $\mathcal{L}^p$ be the set of all possible levelling functions $l : V_p \rightarrow \mathbb{N}$ for the pomset, as defined above. Let $\mathcal{L}_1^p \subset \mathcal{L}^p$ denote the set of all levelization functions l with two extra properties:

3. If $V_1 = \{\bar{e}_i | \nexists \bar{e}_j, \bar{e}_j \leq_p \bar{e}_i\}$ then for at least one $\bar{e} \in V_1$, $l(\bar{e}) = 1$

4. Let $V_p(\bar{e}_j)$ = $\{\bar{e}_{i_1}, \bar{e}_{i_2}, ..., \bar{e}_{i_k}\}$ be the set of events **s.t.** $\forall \bar{e}_i \in V_p(\bar{e}_j)$
if $\bar{e}_i \leq_p \bar{e}_j$, $\nexists \bar{e}_k \notin V_p(\bar{e}_j)$ $(\bar{e}_k \leq_p \bar{e}_j)$, **then** there is at least one e_j **s.t.** $l(\bar{e}_j)$ = $max_{\bar{e}_i \in V_p(\bar{e}_j)}$ $l(\bar{e}_i)$ + 1

This means that this special levelizing function assigns the events with no preceding events a level of 1, and increments by 1.

Intuitively, the levelling function assigns the levels to all the events. Property 1 states that the events that occur late always have higher levels. Property 2 states that functionally dependent events are assigned the highest level of all the events that it is dependent on. Property 3 states that the events in the lowest level have no events preceding them. Finally, property 4 assigns the level numbers to all events.

Example 9. *Consider the traces of variables 'x' and 'z' from example 8. The pre-order $\leq_p$ is defined on events of x as $\bar{x}_1 \leq_p \bar{x}_2 \leq_p \bar{x}_3 \leq_p \bar{x}_4$ and events of z as $\bar{z}_1 \leq_p \bar{z}_2 \leq_p \bar{z}_3 \leq_p \bar{z}_4$. Also, as $z = f(x)$, the functional dependence $\leq_p^f$ is defined as $\bar{x}_i \leq_p^f \bar{z}_i$ for $i = 1, 2, 3, 4$. Property 1 of definition 3 implies that level of $\bar{x}_i$ is less than level of $\bar{x}_{i+1}$. The same applies for the events of z. Now, property 2 implies that events $\bar{x}_i$ and $\bar{z}_i$ are in the same level as $\bar{z}_i$ is functionally dependent on $\bar{x}_i$ ($\bar{x}_i \leq_p^f \bar{z}_i$). Property 3 defines a levelling function which starts by levelling $\bar{x}_1 = 1$, and property 4 describes how to level the others (e.g. $l(\bar{x}_2) = l(\bar{x}_1) + 1 = 2$). This property extends to other events as well as providing the level number for each event. The graphical representation of the pomset and its levelization is shown in Figure 9.4.*

Given a pomset $p : \langle V, \leq, \Sigma, \mu \rangle$, and a levelling function $l \in \mathcal{L}^p$, (p, l) is an $l-$synchronous view of pomset p. A pomset may have multiple $l-$synchronous views. If $l \in \mathcal{L}_1^p$, then we call it a asap-synchronous view of the pomset. Note that "asap" stands for "as soon as possible". One can imagine a levelling function as a schedule for the events of the pomset. In this sense, an asap-synchronous view is provided by a levelling function which makes at least some of the events occur as soon as they are permissible by the constraints of $\leq_p^f$ and $\leq_p$.

An asynchronous view of a pomset is a pomset without a levelling function. Such a view does not show whether non-causal events happen synchronously or not. In an $l-$synchronous view, one can imagine that events that have the same level happen at the same synchronous step. So an $l-$synchronous view imposes a scheduling on the events of the pomset.

Now, if one replaces the clause (1) in extra conditions on $\mathcal{L}_1^p$ by the condition $\forall \bar{e} \in E_1$, $l(\bar{e}) = 1$, then it provides an "exactly-asap" view. We will call the set of such levelling functions as $\mathcal{L}_1^p - exact$. So the corresponding "exactly-asap" schedule, therefore, makes events that results from computing a function on other events, synchronously occurring with the events it depends on. It also makes event sequences in each input to occur in consecutive steps (levels). All events that are inputs (events that do not functionally depend on other events in the pomsets), their sequence happens synchronously in a *lock* step. So an "exactly-asap" view of a pomset is a synchronous scheduling of events, assuming input events all come in the lock step. In Figure 9.5, one can easily see that righthand side we have an exactly-asap-synchronous view of the pomset on the left hand side. The level function l in this view maps $\bar{a}$ and $\bar{b}$ to level 1, and $\bar{c}$ to level 2 and $\bar{d}$ to level 3. Since this pomset does not have an associated $\leq^f$, this level function can be easily seen to be in $\mathcal{L}_1^p$. In Section 3, we give more examples to clarify this idea.

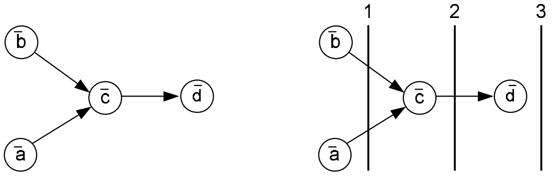


Figure 9.5: Graphical representation of pomset

A process can have multiple behaviors, and can be represented by either a single pomset or by multiple pomsets.

Definition 4. ***Process:*** *A process $\mathcal{P}$ is a collection of pomsets. A process $\mathcal{P} = \{p_1, p_2, \ldots\}$, where $p_1, p_2, \ldots$ are pomsets with disjoint event sets of the process.*

Note that the different pomsets of a process capture different possible behaviors.

Definition 5. ***Pomset Projection:*** *A projection of a pomset p on a variable set I denoted by $p|_I$ is defined as follows. Given p = $\langle V, \leq, \Sigma, \mu \rangle$ and a variable set $I \subseteq \mathcal{V}$, $p|_I$ = $\langle V|_I, \leq |_I, \Sigma|_I, \mu|_I \rangle$, where,*

1. $\Sigma|_I \subseteq \Sigma$ denotes the set of all actions that pertain to variables only in I,
2. $V|_I \subseteq V$ denotes all events that pertain to the variable set I only, ($\forall \bar{e} \in V$, $\bar{e} \in V|_I$ **iff** $\mu(e) \in \Sigma|_I$),
3. $\mu|_I : V|_I \rightarrow \Sigma|_I$ is restriction on μ to $V|_I$, and
4. $\leq |_I \subseteq \leq$ is a restriction of $\leq$ to the events only in V^I.

Intuitively, a restriction of a pomset p on variable set I can be obtained by deleting all events not pertaining to variables in I, and all dependency arrows that shows dependency between such events.

A process $\mathcal{P}$, with input variables I and their corresponding tomsets, is deterministic iff the pomset associated with the process behaviors is unique upto isomorphism.

Definition 6. ***Deterministic Process:*** *A process $\mathcal{P}$ is deterministic* ***iff****, $\exists I \subset \mathcal{V}_{\mathcal{P}}$ s.t $\forall p, q \in \mathcal{P}$,* ***if*** *$p|_I = q|_I$* ***then*** *$p = q$.*

For every given input sequence in the pomset representation of a deterministic process, there exists a single pomset p such that $p|_I$ matches that given input sequence. Note that for an independent input variable set I, its input sequences correspond to a set of tomsets where each tomset represents an event sequence for a variable. For non-deterministic processes, multiple pomsets may represent possible behaviors of the process for the same given input sequences. For the rest of the chapter, we only talk about deterministic processes.

The composition of two pomsets is defined as follows:

Definition 7. ***Composition*** *$(|)$****:*** *Consider two pomsets of x and y as $\langle V_x, \leq_x, \Sigma_x, \mu_x\rangle$ and $\langle V_y, \leq_y, \Sigma_y, \mu_y\rangle$, then composition of x and y is defined as follows:*

$$x \mid y = \langle V_{x|y}, \leq_{x|y}, \Sigma_{x|y}, \mu_{x|y}\rangle, where \begin{cases} V_{x|y} = V_x \bowtie V_y \\ \leq_{x|y} = \leq_x \cup \leq_y \\ \Sigma_{x|y} = \Sigma_x \cup \Sigma_y \\ \mu_{x|y} = \mu_x \cup \mu_y \end{cases}$$

where, $\bowtie$ denotes the unification of the event traces. Let consider $\pi(var)$ to contain all events associated with variable var, and $\mathcal{V}_j$ be the set of variables associated with events $\bar{e} \in V_j$ for $j = x, y$. $\bowtie$ satisfies the following:

1. $\forall \bar{e}_i \in \pi(var)$, **if** $var \in (\mathcal{V}_x \cap \mathcal{V}_y)$ **then** $\bar{e}_i = \bar{e}_{ix} = \bar{e}_{iy}$
2. $\forall \bar{e}_i \in \pi(var)$, **if** $var \in (\mathcal{V}_x \setminus \mathcal{V}_y)$ **then** $\bar{e}_i = \bar{e}_{ix}$
3. $\forall \bar{e}_i \in \pi(var)$, **if** $var \in (\mathcal{V}_y \setminus \mathcal{V}_x)$ **then** $\bar{e}_i = \bar{e}_{iy}$

Event $\bar{e}_{ix}$ and $\bar{e}_{iy}$ are the i^{th} events of pomsets x and y. The composition $(|)$ operator is associative and commutative. When two processes are composed together, the behavior of the resulting process is the combination of the pomsets of the two processes.

Example 10. *Consider two processes with behaviors z:= x default y and s:= z default t. Their corresponding pomsets and composition is shown in figure 9.6.*

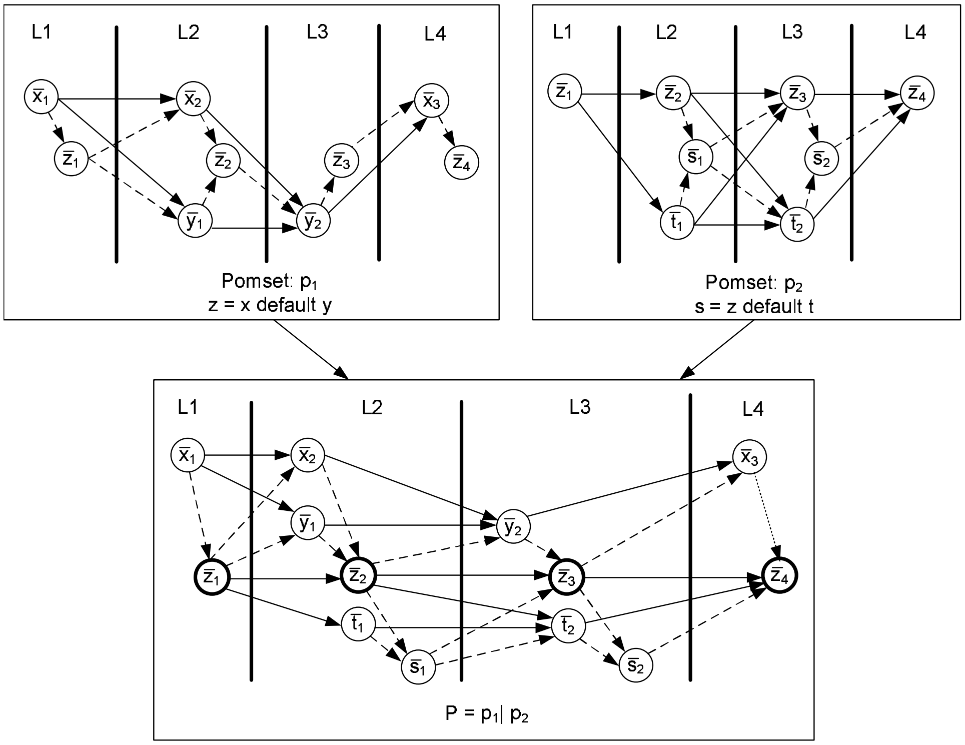


Figure 9.6: Composition of two pomsets.

9.2.1 Flow and Clock Equivalence

The synchronous structure of polychrony is defined by a clock equivalence relation. Two behaviors are clock equivalent if they have the same partial order up to an isomorphic choice of time tags. Note that these definitions are of tagged-based model of polychrony [122], and later we provide extensions to pomsets.

Definition 8. ***Stretching:*** *A behavior c is a* stretching *of b, written $b \ll c$,* ***iff*** $\mathrm{vars}(b) = \mathrm{vars}(c)$ *and there exists a bijection f on $\mathcal{T}$ which satisfies:*

1. $\forall t, t' \in \mathrm{tags}(b), t \leq f(t) \wedge (t < t' \Leftrightarrow f(t) < f(t'))$
2. $\forall x \in \mathrm{vars}(b), \mathrm{tags}(c|_{\{x\}}) = f(\mathrm{tags}(b|_{\{x\}})) \wedge \forall t \in \mathrm{tags}(b|_{\{x\}}), b|_{\{x\}}(t) = c|_{\{x\}}(f(t))$

Definition 9. ***Clock Equivalence:*** *b and c are* clock equivalent, *written $b \sim c$,* ***iff****, there exists d* ***s.t.*** *$b \ll d$ and $c \ll d$.*

$$\left|\begin{array}{l} \forall t, t' \in \mathrm{tags}(b), t \ll f(t) \wedge (t < t' \Leftrightarrow f(t) < f(t')) \\ \forall x \in \mathrm{vars}(b), \mathrm{tags}(c|_{\{x\}}) = f(\mathrm{tags}(b|_{\{x\}})) \wedge \\ \qquad\qquad\qquad \forall t \in \mathrm{tags}(b|_{\{x\}}), b|_{\{x\}}(t) = c|_{\{x\}}(f(t)) \end{array}\right.$$

The asynchronous structure of polychrony is modeled by a flow equivalence relation. Two behaviors are flow equivalent if the order of values they hold is same.

Definition 10. ***Relaxation:*** *A behavior c is a* relaxation *of b, written $b \sqsubseteq c$,* ***iff*** $\text{vars}(b) = \text{vars}(c)$ *and, for all $x \in \text{vars}(b)$, $b|_{\{x\}} \ll c|_{\{x\}}$.*

Definition 11. ***Flow Equivalence:*** *b and c are* flow-equivalent*, written $b \approx c$,* ***iff*** *there exists d* ***s.t.*** *$b \sqsupseteq d$ and $d \sqsubseteq c$.*

Asynchronous composition $r \parallel s$ is defined by considering the partial-order structure induced by the relaxation relation. The parallel composition of r and s consists of behaviors d that relax the behaviors b_r and b_s from r and s along shared signals $I = \text{vars}(r) \cap \text{vars}(s)$ and that stretch b_r and b_s along independent signals of r and s. Let $J = \text{vars}(r) \cup \text{vars}(s)$, then we have
$r \parallel s = \{d \in \mathcal{B}|_J|, \exists (b_r, b_s) \in r \times s, d/I \gg (b_r/I \,\|\, b_s/I) \wedge b_r|_I \sqsubseteq d|_I \sqsupseteq b_s|_I\}$.

We now extend these notions of flow and clock equivalence to pomsets. Let p : $\langle V_p, \leq_p, \Sigma_p, \mu_p \rangle$ and q : $\langle V_q, \leq_q, \Sigma_q, \mu_q \rangle$ be two pomsets and $\mathcal{P}$ a process, such that $p, q \in \mathcal{P}$. We denote the variable set of p and q by $\mathcal{V}_p$ and $\mathcal{V}_q$.

Definition 12. ***Flow map between Pomsets:*** *Given two pomsets p and q with their associated functional dependency ordering $\leq_p^f$ and $\leq_q^f$, a flow map between p and q is an injective map $f_{pq} : V_p \to V_q$* ***s.t.***

1. *$\mathcal{V}_p = \mathcal{V}_q$*
2. *$\forall v \in \mathcal{V}_p, \forall a, b \in V_p|_{\{v\}}$* ***if*** *$a \leq_p b$* ***then*** *$f(a) \leq_q f(b)$, $f(a), f(b) \in V_q|_{\{v\}}$*
3. *$\forall a, b \in V_p$* ***if*** *$a \leq_p^f b$* ***then*** *$f(a) \leq_q^f f(b)$, $f(a), f(b) \in V_q|_{\{v\}}$*

Note that in the above definition $V_p|_{\{v\}}$ denotes only the events in V_p that pertain to a variable v only.

Intuitively, a flow map requires that for every variable in one behavior (pomset), there is a corresponding variable in the other behavior (pomset) where the order of events on these variables is identical. Moreover, if one observes the events across variables that have functional dependence among themselves, such dependence is preserved in the corresponding mapped events.

Definition 13. ***Flow Equivalent Pomsets:*** *Given two pomsets p and q and their associated functional dependency pre-order $\leq_p^f$ and $\leq_q^f$, we call p and q flow equivalent* ***iff*** *there exists a flow map f_{pq} from p to q* ***s.t.*** *f_{pq}^{-1} is also a flow map from q to p.*

Two pomsets (with their functional dependencies) are **flow equivalent** iff by just observing event sequences for each variables separately (without considering their causality with events from other variables, except for functional dependency), one cannot distinguish between the behaviors represented by the two pomsets.

Theorem 13. *For a deterministic process, the flow equivalence classes are singleton sets.*

Proof. (Proof by contradiction) Lets assume that there exists a deterministic process where the flow equivalence classes are not singleton sets. Let $f^1, ..., f^n$ represent flow equivalent classes where for $1 \leq i \leq n$, $1 \leq j \leq n$, and $i \neq j$, $\exists i, j : f^i \neq f^j$ (non-singleton). Hence, f^i and f^j yield different orders of events for the same behavior making it non-deterministic. Hence, it is a contradiction since the process is assumed to be deterministic. □

Let $\mathcal{L}^{\mathcal{P}}$ denote the set of levelling functions for two pomsets in process $\mathcal{P}$, then $\mathcal{L}^p$ and $\mathcal{L}^q$ denote the sets of levelling functions for pomsets p and q. Two pomset are clock equivalent if they have isomorphic levels.

Definition 14. ***Clock Equivalent Pomsets:*** *Given two pomsets $p = \langle V_p, \leq_p, \Sigma_p, \mu_p \rangle$ and $q = \langle V_q, \leq_q, \Sigma_q, \mu_q \rangle$ and their associated functional dependencies $\leq^f_p$ and $\leq^f_q$, we say p and q are clock equivalent, denoted as $p \sim q$, **iff** there exists a bijection $f : V_p \rightarrow V_q$, and levelling functions $l_p \in \mathcal{L}^p_1$ and $l_q \in \mathcal{L}^p_1$ **s.t.** $\forall x \in V_p$, $l_p(x) = l_q(f(x)) \wedge \forall y \in V_q$, $l_q(y) = l_p(f^{-1}(y))$.*

What this means is that two pomsets are **clock equivalent** if asap-synchronous views of both are created, then a bijection between the events in both would exist with the following property. Events in one pomset that are in the same level under its levelling function, will be mapped to events in the other in a way that they all have the same level under that pomset's levelling function. In other words, if one thinks of events in the same level happening in the same cycle, then in both pomsets events that correspond to each other would happen in the same cycle.

9.2.2 Understanding Endochrony

A pomset represents a single behavior of a polychronous program for a given input event sequence. So from an asynchronous observer's point of view, a pomset provides a specification such that his/her observations of event sequences (without looking into the program itself) must never violate the pomset's $\leq$ and $\leq^f$. On the other hand, a levelling function l provides a view of the program execution with the scheduling of the events, which means the observer can also see inside the program's execution schedule with respect to some clock. Each possible levelling function may correspond to a different clock, and schedule. In polychrony, an interesting class of processes/programs are called endochronous processes/programs. This is the class of programs that can be scheduled with one master clock uniquely.

Definition 15. *[122] A process $\mathcal{P}$ is endochronous **iff**, $\forall b_i, b_j$ behaviors of $\mathcal{P}$, **if** $b_i \approx b_j$ **then** $b_i \sim b_j$*

What this means is that if we have behaviors that are flow equivalent, we could schedule them uniquely with respect to one clock. The way to schedule them will be given by the $\mathcal{L}_1$ levelling function set for the behaviors.

Now we state the following theorem which demystifies the idea of endochrony.

Theorem 14. *A process is endochronous* ***iff****,* $\forall p \in \mathcal{P}$ *the set* $\mathcal{L}_1^p$ *is singleton.*

Proof. Consider any behaviors b_i, b_j for an endochronou process p, and their corresponding synchronous and asynchronous views: $\mathcal{L}_1^p$.

We have the following:

1. **P1:** Property of Endochrony: $b_i \approx b_j \Rightarrow b_i \sim b_j$
2. **P2:** Pomset view: singleton $\mathcal{L}_1^p$

P1 states that in an endochronous process p, all flow equivalent behaviors (modeled as flow equivalent pomsets) are also clock equivalent (modeled as clock equivalent pomsets). On the other hand, **P2** states that the equivalent classes of $\mathcal{L}_1^p$ are singleton (synchronous view). We need to show that **P1** $\Leftrightarrow$ **P2**. So, we prove that (i) **P1** $\Rightarrow$ **P2**, and (ii) **P1** $\Leftarrow$ **P2**. Now for proving (i), we know that $b_i \approx b_j \Rightarrow b_i \sim b_j$, and by definition of clock equivalence, $b_i \sim b_j \Rightarrow b_i \approx b_j$. Hence, the clock equivalence class is singleton. Therefore, the set $\mathcal{L}_1^p$ will contain a unique synchronous view, and would be singleton. Hence we have proved (i).

For (ii), we know $\mathcal{L}_1^p$ is singleton implying that we have a singleton clock equivalent class of behaviors $(b_i \sim b_j)$. Therefore, $singleton\ \mathcal{L}_1^p \Leftrightarrow b_i \sim b_j$ (Singleton clock equivalence class implies a unique synchronous view). Now, we know by definition of clock equivalence that $b_i \sim b_j \Rightarrow b_i \approx b_j$. Lets consider $\mathcal{L}_1^p - exact$ of the same behaviors (asynchronous view). So, $singleton\ \mathcal{L}_1^p \Leftrightarrow \mathcal{L}_1^p - exact$. Also, by definition of asynchronous views, and flow equivalence, for the same set of behaviors, $\mathcal{L}_1^p - exact \Leftrightarrow b_i \approx b_j$. Hence, we get $b_i \sim b_j \Leftrightarrow b_i \approx b_j$. Therefore, we can conclude that singleton $\mathcal{L}_1^p \Rightarrow b_i \approx b_j \Rightarrow b_i \sim b_j$

Hence, we have proved (ii). We can conclude that $\mathbf{P1} \Leftrightarrow \mathbf{P2}$.

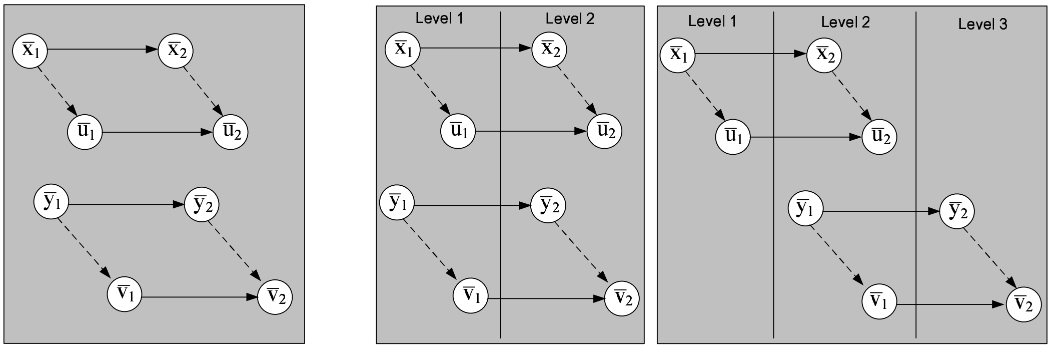
□

We provide examples to clarify the understanding of endochrony based on our pomset based semantics. Consider a simple SIGNAL program shown in listing 9.1. The inputs and outputs are declared in line 1, where inputs are labelled with '?' and outputs with '!'. In the program, x, y are inputs which are assigned to outputs u, v respectively. The clocks of x and u are same, and clock of y and v are same because of the assignment. However, the clocks or x and y are unrelated, implying that the clocks of u and v are unrelated.

Listing 9.1: Non-endochronous SIGNAL program

```
1 process P = (? integer x, y; ! integer u, v;)
2             (| u:= x
3              | v:= y
4              |)
```

Consider, two input events on signals x and y each, represented as x_1, x_2, and y_1, y_2. The pomset representation of these behaviors are shown in figure 9.7. Note that the solid arrow represents the causal dependence, whereas the dotted arrow represents the functional dependence.



(a) Asynchronous View (b) Multiple Synchronous Views

Figure 9.7: Pomset representation of a non-endochronous program

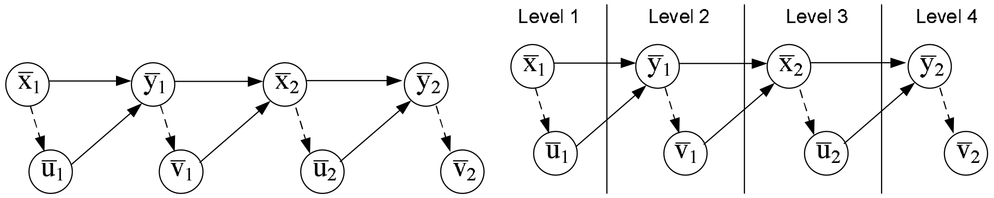
Figure 9.7(a) shows the asynchronous view of the pomset. Now, levelizing the pomset, we can get multiple synchronous views as shown in figure 9.7(b). Recall that the causally dependent events are placed in different levels, whereas functional dependent events are placed in the same level. We only show two synchronous views, however, many different can be observed.

Next, we try to transform this program to an endochronous program by relating the clocks of x and y. Consider the modified program shown in listing 9.2.

Listing 9.2: Endochronous SIGNAL program

```
process P = ( ? integer x, y; boolean cx, cy; ! integer u, v;)
  (| u:= x
   | v:= y
   | cx ^= cy
   | x ^= when cx
   | y ^= when cy
   |)
```

The non-endochronous program (listing 9.1) is extended by adding boolean signals and clock constraints to make it endochronous (listing 9.2). The clocks of cx and cy are identical, and the clock is x is present whenever the cx is present and true. The same is done for signal y. The pomset view now depends on the clock of cx, cy, and their corresponding values. Hence, given these values, we always get a unique synchronous view. If the boolean true values of cx and cy alternate, the corresponding pomset with asynchronous and synchronous views is shown in Figure 9.8. For clarity, we do not show the events of signals cx and cy.



(a) Asynchronous View (b) Synchronous View

Figure 9.8: Pomset represention of an endochronous program

9.3 Constructing Pomset Semantics for Polychronous SIGNAL

The tagged model of polychrony consists of behaviors, where each behavior corresponds to a signal. A signal in this model is a sequence of (values, tag) pairs. As discussed earlier, each signal corresponds to a unique variable, and functionally dependent signals are created based on the corresponding operations. We use a running example to show this construction.

Example 11. *Consider the following example with two variables 'a' and 'b' and their respective traces. Some operation $c = a \oplus b$ is defined such that whenever an event for variable 'b' is present, the new behavior 'c' gets the value of 'a'.*

$time:$	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
$a:$	1	-	2	-	3	-	4	-	5
$b:$	1	2	-	3	4	-	5	6	-
$c = a \oplus b:$	1	-	-	-	3	-	4	-	-

The events on variable a are: $(1, t_1)$, $(2, t_3)$, $(3, t_5)$, $(4, t_7)$, $(5, t_9)$, and b are: $(1, t_1)$, $(2, t_2)$, $(3, t_4)$, $(4, t_5)$, $(5, t_7)$, $(6, t_8)$. We represent these events as a_1, a_2, a_3, a_4, and a_5 for variable a, and b_1, b_2, b_3, b_4, b_5, and b_6 for variable b. For this example, we define a new behavior c which is formed by doing operation $\oplus$ of the events on a and b. The operation $\oplus$ requires both the events to be present. Hence, if there exists a matching tag for the events of a and b, then an event for variable c is created. This can be seen from the trace of variable c in the example.

As discussed earlier, there is a difference between the events in tagged-based model and pomset based model. Furthermore, the partial ordering relation has to be captured based on these events.

To construct its corresponding pomset semantics, we do the following:

1. Create appropriate events for pomsets.
2. Arrange these new events based on an ordering relation.

Each tagged event is transformed to a pomset event. Each tagged event e for the variable (var) of the form $e = (value, tag)$ is transformed into a pomset event $\bar{e}$ of the form $\bar{e} = (var, value)$.

Hence, for all events of the tagged-based model, we can create the set of vertices (modeling events) V. Also, we know that since the tags are partially ordered, we can create the partial order $\leq$ for the events of the pomset. Σ consists of the actions, and the labelling function μ assigns the actions to the vertices.

For our example, we can create the corresponding events for the pomset. The events for variable a are $\bar{a}_1$,$\bar{a}_2$,$\bar{a}_3$,$\bar{a}_4$, and $\bar{a}_5$, and for variable b are $\bar{b}_1$,$\bar{b}_2$,$\bar{b}_3$,$\bar{b}_4$,$\bar{b}_5$ and $\bar{b}_6$. The corresponding partial ordering between these events is defined by $\leq$. ‘1’ of Figure 9.9 shows the corresponding pomset representation of the behaviors of a and b. Application of the levelling function give the levels for the events.

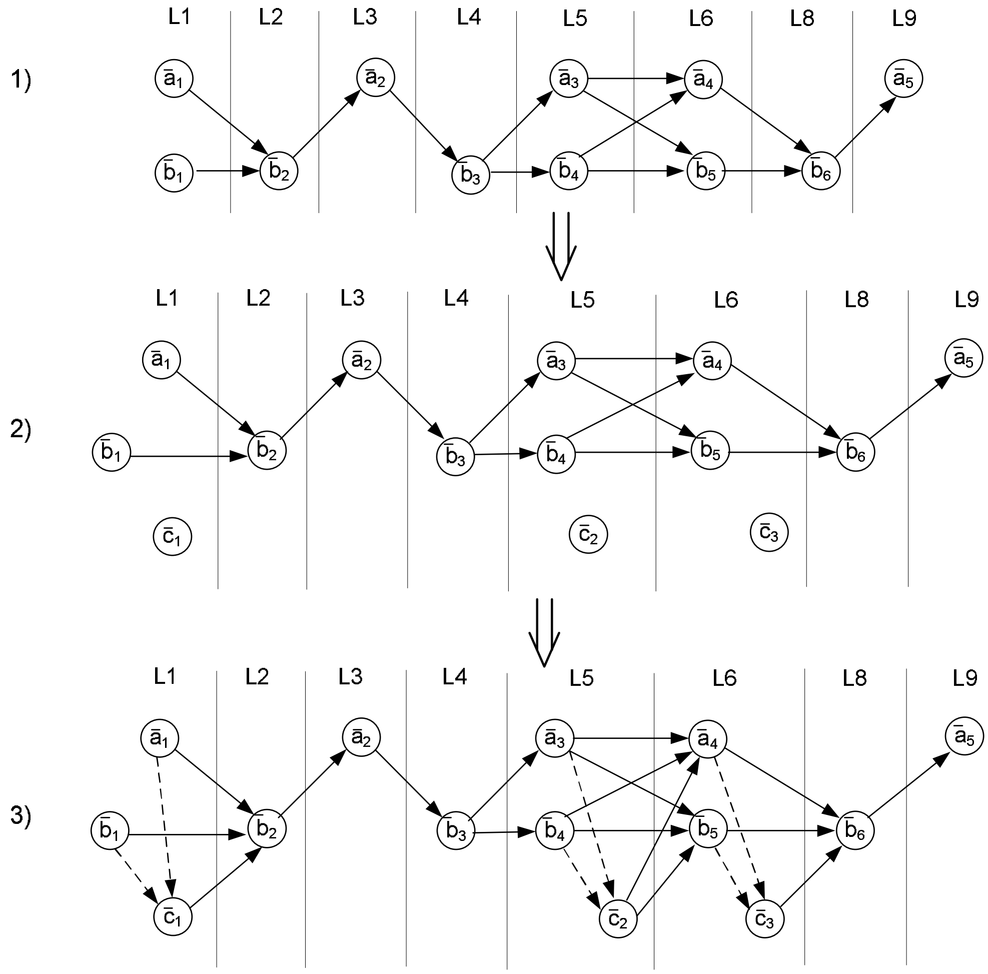


Figure 9.9: Levelized graphical representation of pomset with events of variable a and b

For each level in figure 9.9, if events for both variables a and b exist, then a resulting event for variable c is created. For instance, in level $L1$, both a_1 and b_1 are present, therefore we get $c_1 = a_1 \oplus b_1$, but in level $L2$, no event results for variable c as the event for variable a is missing (‘2’ of figure 9.9). After we evaluate all the levels, the functional dependent ordering ($\leq^f$) is constructed. For example, in level 1, we create $a_1 \hookrightarrow c_1$ and $b_1 \hookrightarrow c_1$. Futhermore, we also create $c_1 \hookrightarrow b_2$ as the next event for both a_1 and b_1 is b_2. The final pomset representation for the interaction is shown in '3' of figure 9.9.

Given this representation, we can analyze the synchronous as well as asynchronous representation of the behavior. Recall that the synchronous representation shows the levelling details of the events, whereas the asynchronous representation shows the "exactly-asap" view. We can construct $\mathcal{L}_1^p$ (asap-synchronous view) as well as $\mathcal{L}_1^p - exact$ (exactly-asap view) for the behavior. Assuming that the synchronous behavior ($\mathcal{L}_1^p$) is singleton, and its coincides with its asynchronous behavior ($\mathcal{L}_1^p - exact$), then this behavior is endochronous (by theorem 14).

9.4 Multi-threading Example

In this section, we explore a different kind of example where we consider event sequences not as results of actions on variables, but event sequences as synchronization events of various threads in a multi-threaded program. We show how the pomset based semantic model can be helpful in understanding the idea of endochrony for such programs.

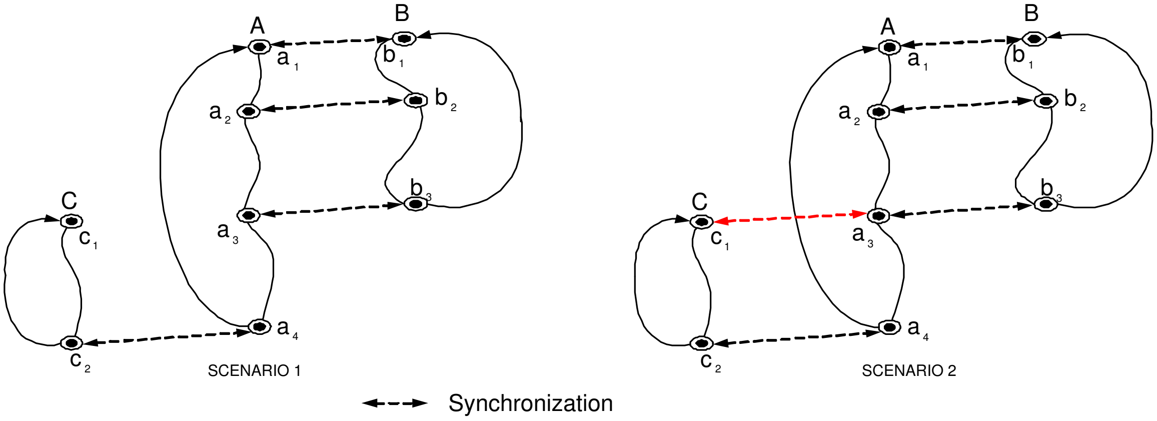


Figure 9.10: Multithreaded model

Consider a multithreaded program consisting of three different threads **A**, **B** and **C** synchronizing with each other. In Scenario 1 of Figure 9.10, thread **A** synchronizes with thread **B** at three different points and with thread **C** at one point during a single execution. The execution trace of **A** begins at a_1 where it synchronizes with point b_1 of thread **B**. The threads continue execution after synchronization at (a_1, b_1), until they re-synchronize at (a_2, b_2) and then at (a_3, b_3). Thread **B** after synchronization at (a_3, b_3) completes one execution and returns to its initial point to begin its next round of execution. Thread **A** continues and further synchronizes with thread **C** at point (a_4, c_2), after which it completes and loops back. The point to note as well as the distinction between the

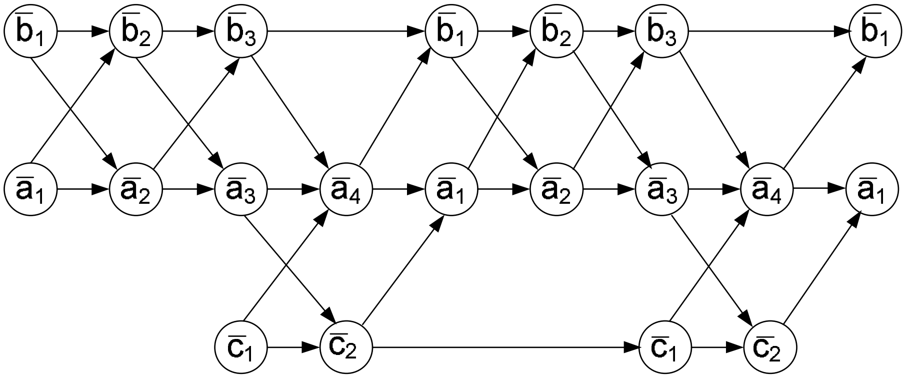


Figure 9.11: Pomset representation of Scenario 1

two scenarios is that thread **C** begin execution at point c_1, which is not synchronized with either of the threads. In Scenario 2, thread **C**'s beginning is synchronized with point a_3 of thread **A**, which in turn with b_3 of thread **B**.

The pomset representation for the multithreaded model of Scenario 1 is shown in Figure 9.11. The pomset events such as a_1, a_2, a_3 and a_4 correspond to the synchronization points of thread **A** and similarly the other events correspond to thread **B** and **C**. The synchronization of the different threads dictate scheduling relations.

Consider the event c_1, the only ordering captured by the pomset is the causal relation $c_1 \leq c_2$. This is because in Scenario 1, thread C does not synchronize with any of the other thread at point c_1. Therefore, no scheduling relations are captured by the pomset. However, the thread synchronizes with **A** at point (a_4, c_2). Therefore, the scheduling relation $c_1 \leq a_4$ is also added to the pomset. But this does not restrict c_1, as it can concurrently execute with any event preceding a_4. Therefore, $l(c_1)$ = 1, 2 or 3 and the pomset has multiple $\mathcal{L}_1$ levelling functions.

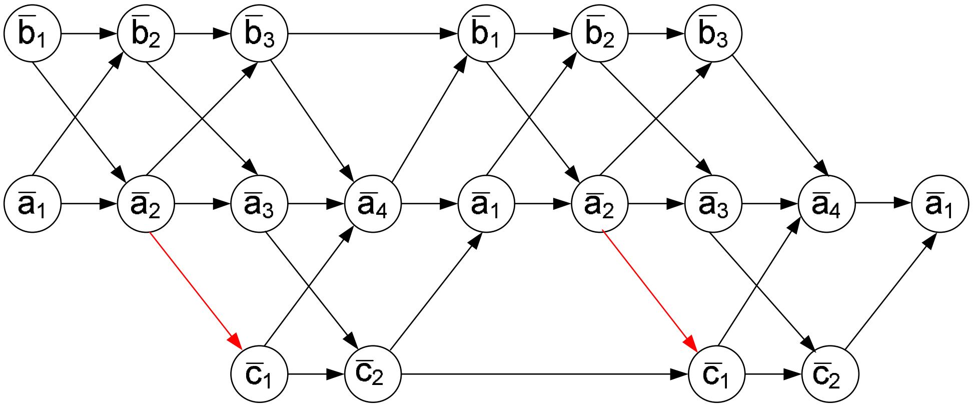


Figure 9.12: Pomset representation of Scenario 2

Consider Scenario 2, where thread **C**, synchronizes with thread A at points (a_3, c_1) and (a_4, c_2).

This restricts the concurrent execution of thread C. In the corresponding pomset shown in Figure 9.12, an additional scheduling relation $a_2 \leq c_1$ will be inserted, which restrict c_1 to the same schedule as a_3. Therefore, Scenario 2 $\mathcal{L}_1$ function set is a singleton. This means that the multi-threaded program in this scenario is endochronous, and there is a unique way of scheduling the threads as far as the synchronization events are concerned. This of course, makes verification of properties of the program a bit easier.

Chapter 10

Conclusion and Future Work

This dissertation focuses on various timing related issues in the context of composition of IPs. Composition of IPs necessitates a refinement based design flow. In such a design flow, a synchronous model [135] of the system can be built where all interconnect latencies are assumed to be negligible. This follows from the synchrony hypothesis used in clock-synchronous hardware design, where, computation and communication latencies are negligible. From this synchronous model, necessary refinements are made to the design to render the design latency insensitive (LI). The correctness criteria is that the LI design be *latency equivalent* to the synchronous design. Two signals are said to be latency equivalent if the sequence of valid or *informative* events on the two signals are identical. In other words, if an observer observes the order of events on two signals, while discounting ‘empty’ events, the two signals will look the same. Since processes can be thought of as consumer and producer of events on signals, the same notion can be extended to systems. Two system models are latency equivalent if their outputs are latency equivalent given both are subjected to the same (or latency equivalent) input sequences.

Although, a number of protocols, including latency insensitive protocols (LIPs) have been published in the literature, no formal framework had been created to validate these protocols. The reason why one needs a framework where variants of these protocols can be quickly validated either through simulation or through model checking is as follows: These protocols are going through a continuing evolution phase. For example, [72] attempts to optimize and improve the protocols described in [71, 68, 69, 70] to obtain a more efficient and simpler protocol circuitry. Although [70] offers a mathematical proof of correctness of their version of LIPs, when optimizations or extensions are made in the subsequent works, no such formal proof is usually offered. Due to the subtleties involved in the optimizations, it is plausible that the newly invented LIPs have serious flaws. We have experienced this in our attempts to optimize Carloni’s protocol [68, 69]. As a result, we felt that there is a need for a framework where these protocols can be quickly modeled and validated. In order to formally verify such protocols, the LI system as well as the synchronous idealization have to be modeled formally, and the latency equivalence has to be captured as a formal property. The best way is to provide designers with an easy to use framework to model and validate their protocols. In this thesis, we provide a validation framework for families of latency insensitive protocols. We present validation techniques and frameworks for ensuring correctness of latency insensitive designs [206, 207, 214]. In the framework, we model the latency insensitive system along with its synchronous idealization and provide the same input signals to both the systems. These input signals can also be latency equivalent. The corresponding outputs of the two systems are then checked to be latency equivalent. Furthermore, we have used our validation framework to verify protocols developed in the industry (elastic controllers). We also present two new latency insensitive protocols: relay-station based protocol and bridge based protocol. Our relay station based protocol is a simplified version of the original protocol proposed in [70]. Such types of protocols involve partitioning of long interconnects. In such approaches, the relay stations are required to be placed at strategic points to ensure that the signal propagation of each wire segment is less than the clock period. This requires multiple place and route iterations. Our bridge based approach replaces these relay stations with additional wires, hence this reduces the number of such place and route iterations.

The proposed LIP approaches require prior knowledge of the latency on the long interconnects. However, such information on the long interconnects and the number of cycles required for data to communicate through them cannot be known until after the final layout of the design. A solution to such a problem is to enable handshaking between synchronous IP blocks. This is the design idealogy of globally asynchronous locally synchronous designs (GALS). However, there is a lack of tools and design methodologies to facilitate GALS designs. In most cases, GALS designs are constructed using ad hoc methods, where synchronous components are encapsulated with some wrapper logic and communication is either handshake-driven or bounded FIFOs are used. Absence of validation tools for such designs make it difficult to ensure correctness. We present a trace-based formal framework that helps in establishing the criteria for correct-by-construction transformation to GALS from a synchronous design. We establish “single-activation” property, and theoretically show that a system with such a property can be transformed to a correct-by-construction GALS using a barrier synchronization protocol. We also present a corresponding barrier synchronization protocol for GALS design. So now a designer can create designs and verify them using available designs tools, and using the protocol presented, the designs that possess “single-activation” in our framework can be transformed to GALS. These designs would be correct by construction, and hence would not require GALS validation tools. We also present a formal refinement based framework which takes us from a kahn process network (KPN) specification towards a GALS implementation. We consider four different GALS architectures, and provide the formal definitions for each of these. The architectures we consider involve different communication protocols. In the *handshake-based architecture*, the communication between processes occur based on well known four-phase handshake protocol, whereas in the *fifo-based architecture*, the communication protocol borrows ideas from latency insensitive protocols with a fifo interface connecting the processes running on different clocks. The *controller-based architecture* consists of a centralized controller that governs the execution of the processes. Finally, we have the *lookup-based architecture* that avoids the signaling based protocols, and the communication happens based on fast data accesses from a lookup storage which is located on-chip. The architectures we present, consist of processes that are continuous and monotonic, and are also correct-by-construction. We show this by transforming the processes into deterministic (I/O) automata, since they have been shown to be continuous and monotonic [156]. Our assumption of starting with KPN processes that execute when all inputs are present and produce output to all output signals, ensures absence of artificial deadlocks. The formalization presented is also necessary to prove the correctness as well as for unambiguously presenting our protocols for GALS realization. We show behavioral equivalence of the KPN and the refined architecture by ensuring that the behaviors of both are latency equivalent.

In context of polychrony, we present a new semantic model that not only provides intuitive characterization of determinacy, and endochrony, but also helps in analyzing the synchronous and asynchronous interpretation of the same program in a single representation. The existing model of polychrony uses a tag-based representation that makes asynchronous interpretation of programs unduly complicated, resulting in difficult to understand theories for characterizing polychronous programs, such as characterization of endochronous programs [42, 41, 122]. In this paper, we bring forth the pomset based semantic model which is intuitive for engineers while it removes the need for a tagged-signal model.

Future Work

There are several opportunities for improvement on the work presented in this dissertation. With respect to the along interconnect problems, several LIPs have been proposed (as discussed in the thesis), however, estimation of the interconnect latencies cannot be made until the final layout of the design. Secondly, lack of verification frameworks makes it difficult to ensure correctness of such protocols. We have presented such a framework in this thesis. Application of such frameworks is needed in different domains to ensure correctness of LIPs.

As discussed earlier, lack of design tools for GALS is forcing the designers to think in terms of fully synchronous designs. If GALS designs are created, they are done in an ad hoc manner, which may be prone to deadlocks or incorrect protocols. Therefore, design frameworks and tools are needed for GALS which will provide a complete framework including modeling, verfication, and synthesis capabilities. Furthermore, efficient implementation components need to be developed for the GALS protocols presented in this thesis, to allow integration of IPs. Other asynchronous communication protocols can also be employed for their communication.

On our work pomset-based semantics for Polychrony, we show how to analyze multi-clock behaviors of Polychronous programs using pomsets. Future work involves completion of the design framework that includes a tool facilitating modeling of designs, analyzing hardware as well as software aspects of the design, simulation, verification and implementation of designs. The tool should be capable of generation code for software as well as hardware part of the embedded design.

Bibliography

[1] "Amba system architecture," http://www.arm.com/products/solutions/AMBAHomePage.html.

[2] "Arm: The architecture for the digital world," http://www.arm.com.

[3] "Bell Labs," http://www.bell-labs.com/.

[4] "Cadence," http://www.cadence.com.

[5] "CoreConnect Bus Architecture," http://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture.

[6] "The gnu m4 macro preprocessor." http://www.gnu.org/software/m4.

[7] "HOL: The High Order Logic Theorem Prover," http://cs.anu.edu.au/student/comp8033/hol.html.

[8] "Ibm," http://www.ibm.com.

[9] "Liberty simulation environment," http://liberty.princeton.edu/LSE.

[10] "LTL 2 BA : fast algorithm from LTL to buchi automata," http://www.liafa.jussieu.fr/˜oddoux/ltl2ba/.

[11] "Mentor graphics," http://www.mentor.com.

[12] "Ocp," http://www.ocpip.org.

[13] "Open SystemC Consortium Initiative," http://www.systemc.org, pp. 1–100.

[14] "Phillips," http://www.nxp.com.

[15] "The polychrony toolset," http://www.irisa.fr/espresso/Polychrony. [Online]. Available: http://www.irisa.fr/espresso/Polychrony

[16] "Software System Award," http://www.acm.org/awards/ssaward.html.

[17] "Sonics inc." http://www.sonicsinc.com.

[18] "SPIN root," http://spinroot.com/spin/whatispin.html.

[19] “The spirit consortium,” http://www.spiritconsortium.org.

[20] “Standard ML of New Jersey,” http://www.smlnj.org.

[21] “Synopsys,” http://www.synopsys.com.

[22] “Texas instruments,” http://www.ti.com.

[23] “The PVS Specification and Verification System,” http://pvs.csl.sri.com/.

[24] “UPPAAL,” http://www.uppaal.com.

[25] “Vsi alliance,” http://www.vsi.org.

[26] “Z/EVES,” http://www.ora.on.ca/z-eves/.

[27] P. Abdulla, P. Bjesse, and N. Een, “Symbolic Reachability Analysis based on SAT-solvers,” in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2000.

[28] W. Acherman and J. Dennis, “VAL-Oriented Algorithmic Language, Preliminary Reference Manual,” MIT, Tech. Rep. MIT-LCS-TR-218, 1979.

[29] A. Agiwal and M. Singh, “An Architecture and a Wrapper Synthesis for Multi-Clock Latency-Insensitive Systems,” in *Proceedings of International Conference on Computer Aided Design (ICCAD-05)*, 2005.

[30] ——, “Multi-Clock Latency-Insensitive Architecture and Wrapper Synthesis,” in *Proceedings of Formal Methods for GALS Systems (FMGALS-05)*, 2005.

[31] T. Amagbegnon, L. Besnard, and P. L. Guernic, “Arborescent canonical form of boolean expressions,” INRIA Research Report, Tech. Rep. 2290, 1994.

[32] ——, “Implementation of the data-flow synchronous language signal,” in *Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI’95)*, 1995.

[33] K. Ara and K. Suzuki, “A Proposal for Transaction-Level Verification with Component Wrapper Language,” in *Proceedings of Design, Automation and Test in Europe (DATE)*, 2003.

[34] Arvind and S. Brobst, “The Evolution of Dataflow Architectures: from Static Dataflow to P-RISC,” *International Journal of High Speed Computing*, vol. 5, no. 2, 1993, world Scientific Publishing Company.

[35] Arvind and R. Nikhil, “Executing a Program on the MIT Tagged-Token Dataflow Architecture,” *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 300–318, March 1990.

[36] D. C. Arvind, “Dataflow architectures,” *Annual review of computer science*, vol. 1, 1986.

[37] M. Azam, P. Franzon, and W. Liu, “Low power data processing by elimination of redundant computations,” in *International symposium on Low power electronics and design*, Monterey, California, United States, August 1997, pp. 259–264.

[38] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, “Efficient detection of vacuity in temporal model checking,” in *Formal Methods of System Design*, March 2001, pp. 141–163.

[39] P. Beerel and T. Meng, “Automatic gate-level synthesis of speed-independent circuits,” in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 1992.

[40] L. Benini and G. D. Micheli, “Networks on Chips: A New SoC Paradigm,” *Computer*, vol. 35, no. 1, pp. 70–78, Jan 2002.

[41] A. Benveniste, B. Caillaud, and P. L. Guernic, “From synchrony to asynchrony,” in *Proceedings of Concurrency Theory*, vol. 1664. Springer, 1999.

[42] A. Benveniste, B. Caillaud, and P. Le Guernic, “Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation,” *Information and Computation*, vol. 163, no. 1, pp. 125–171, 2000.

[43] A. Benveniste, D. Caillaud, L. Carloni, and A. Sangiovanni-Vincentelli, “Tag Machines,” in *Proceedings of Embedded Software Conference, Lecture Notes in Computer Science, Springer Verlag*, October, 2005.

[44] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, “The synchronous languages 12 years later,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[45] A. Benveniste, P. Caspi, P. L. Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis, “A protocol for loosely time-triggered architectures.” in *Proceedings of Embedded Systems Software (EMSOFT)*, Springer, Ed., vol. 2491, 2002, pp. 252–266.

[46] A. Benveniste, P. Caspi, M. Natale, C. Pinello, A. Sangiovanni-Vincentelli, and S. Tripakis, “Loosely Time-Triggered Architectures based on Communication-by-Sampling,” INRIA-IRISA, Tech. Rep. 1854, 2007.

[47] A. Benveniste, P. Bournai, T. Gautier, M. L. Borgne, P. L. Guernic, and H. Marchand, “The Signal declarative synchronous language: controller synthesis & systems/architecture design,” *Proceedings of 40th IEEE Conference on Decision and Control*, 2001.

[48] S. Berezin, A. Biere, E. Clarke, and Y. Zhu, “Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-Order Processor Verification.” in *Proceedings of Formal Methods on Computer-Aided Design*, 1998.

[49] R. Bergamaschi and W. Lee, "Designing systems-on-chip using cores," in *Proceedings of Design Automation Conference*, 2000.

[50] K. Berkel, "Beware the Isochronic Fork," *Integration, the VLSI Journal*, vol. 13, no. 2, pp. 103–128, 1992.

[51] ——, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 2004, no. 5.

[52] K. Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij, and R. Wiel, "A Single-Rail Re-implementation of a DCC Error Detector using a Generic Standard-Cell Library," in *Proceedings of Asynchronous Design Methodologies*, 1995, pp. 72–79.

[53] K. Berkel and R. Burgess, "Asynchronous circuits for low power," *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 22–32, 1994.

[54] K. Berkel, M. Joseph, and S. Nowick, "Scanning the Technology: Applications of Asynchronous Circuits," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 223–233, 1999.

[55] G. Berry and E. Sentovich, "Multiclock Esterel," in *Proceedings of Correct Hardware Design and Verification Methods (CHARME)*, 2001.

[56] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[57] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT Procedures Instead of BDDs," in *Proceedings of Design Automation Conference*, 1999.

[58] M. T. Bohr, "Interconnect scaling - The real limiter to high performance VLSI," *In IEEE Int. Electron Devices Meeting*, pp. 241–244, 1995.

[59] J. Boucaron, J.-V.Millo, and R. Simone, "Formal Methods for Scheduling of Latency Insensitive Designs," Institut National De Recherche En Informatique Et En Automatique (INRIA), Tech. Rep., 2007.

[60] J. Boucaron and J.-V. Millo, "Compositionality of Statically Scheduled IP," in *Proceedings of Third International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Systems.*, 2007.

[61] A. Branover, R. Kol, and R. Ginosar, "Asynchronous Design By Conversion: Converting Synchronous Circuits into Asynchronous Circuits," in *Proceedings of Design Automation and Test in Europe (DATE)*, 2004.

[62] D. Brooks and M. Martonosi, "Value-based Clock Gating and Operation Packing: Dynamic Strategies for Improving Processor Power and Performance," *ACM Trans. Comput. Syst.*, vol. 18, no. 2, pp. 89–126, 2000.

[63] J. Brzozowski and J. Ebergen, “On the Delay-Sensitivity of Gate Networks,” *IEEE Transactions on Computers*, vol. 41, no. 11, pp. 1349–1360, 1992.

[64] R. J. A. Buhr, *System Design with ADA*. Prentice-Hall, 1984.

[65] J. Burch and D. Dill, “Automatic verification of pipelined microprocessor control.” in *Proceedings of Computer-Aided Verification (CAV)*, 1994.

[66] S. Burns, “Performance Analysis and Optimization of Asynchronous Circuits,” Ph.D. dissertation, California Institute of Technology, 1991.

[67] L. Carloni, “The Role of Back-Pressure in Implementing Latency-Insensitive Systems,” *Electronic Notes in Theoretical Computer Science*, vol. 146, pp. 61–80, 2006.

[68] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, “A Methodology for Correct-by-Construction Latency Insensitive Design,” in *Proceedings of International Conf. Computer Aided Verification*, November 1999, pp. 309–315.

[69] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, “Latency Insensitive Protocols,” in *Proceedings of International Conference on Computer-Aided Verification (ICCAD)*, vol. 1633. Trento, Italy: Springer Verlag, 07 1999, pp. 123–133.

[70] ——, “The Theory of Latency Insensitive Design,” *In IEEE Transactions on Computer Aided Design of Integrated Circuits and System*, vol. 20, no. 9, pp. 1059–1076, 2001.

[71] L. Carloni and A. Sangiovanni-Vincentelli, “Coping with Latency in SoC Design,” *In IEEE Micro, Special Issue on Systems on Chip*, vol. 22, no. 5, p. 12, October 2002.

[72] M. Casu and L. Macchiarulo, “A New Approach to Latency Insensitive Design,” in *Proceedings of Design Automation Conference (DAC)*, 2004.

[73] ——, “On-chip transparent wire pipelining,” in *Proceedings of International Conference on Computer Design (ICCD)*, 2004.

[74] A. Chakraborty, “Efficient Self-Timed Interfaces for Crossing Clock Domains,” Master’s thesis, The University of British Columbia, 2003.

[75] A. Chakraborty and M. Greeenstreet, “Efficient Self-Timed Interfaces for Crossing Clock Domains,” in *Proceedings of International Symposium on Asynchronous Circuits and Systems*, 2003.

[76] A. Chakraborty and M. Greenstreet, “A Minimal Source-Synchronous Interface,” in *Proceedings of ASIC/SoC Conference*, 2002.

[77] S. Chakraborty, J. Mekie, and D. Sharma, “Reasoning about Synchronization in GALS Systems,” *Formal Methods in System Design*, vol. 28, no. 2, pp. 153–169, 2006.

[78] T. Chelcea and S. Nowick, “Low-Latency Asynchronous FIFOs using Token Rings,” in *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.

[79] ——, “Robust Interfaces for Mixed-Timing Systems,” *IEEE Transactions on VLSI*, vol. 12, no. 8, pp. 857–873, Aug 2004.

[80] C. Chou and D. Peled, “Formal Verification of a Partial Order Reduction Technique for Model Checking,” in *Automated Reasoning*, ser. 3, November 1999, vol. 23, pp. 265–298.

[81] T. Chu, C. Leung, and T. Wanuga, “A Design Methodology for Concurrent VLSI Systems,” in *Proceedings of International Conference on Computer Design (ICCD)*, 1985.

[82] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “NuSMV: A New Symbolic Model Verifier,” *Lecture Notes in Computer Science*, vol. 1633, pp. 495–499, 1999.

[83] K. Claessen, R. Hahnle, and J. Martensson, “Verification of Hardware with First-Order Logic,” in *Proceedings of PaPS*, 2002.

[84] E. Clarke, E. Emerson, and A. Sistla, *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*, ser. Issue 2. ACM Transactions On Programming Languages and Systems, April 1986, vol. Vol. 8, pp. 244–263.

[85] E. M. Clarke and E. A. Emerson, “Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic,” in *Proceedings of the Workshop on Logic of Programs*, vol. 131. Yorktown Heights, New York: Springer-Verlag, May 1981, pp. 52–71.

[86] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. The MIT Press, 1999.

[87] B. Cohen, *VHDL Coding Styles and Methodologies*. Kluwer, 1999.

[88] J. Cortadella, M. Kishinevsky, and B. Grundmann, “Specification and design of synchronous elastic circuits,” in *Proceedings of International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, 2006.

[89] ——, “Synthesis of Synchronous Elastic Architectures,” in *Proceedings of ACM/IEEE Design Automation Conference*, 2006.

[90] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, “Desynchronization: Synthesis of Asynchronous Circuits from Synchronous Specifications,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, 2006.

[91] A. Davare, K. Lwin, A. Kondratyev, and A. Sangiovanni-Vincentelli, “The Best of Both Worlds: The Efficient Asynchronous Implementation of Synchronous Specifications,” in *Proceedings of Design Automation Conference (DAC)*, 2004.

[92] A. Davis, "The Architecture and System Method of DDM-1: A Recursively-Structured Data Driven Machine," in *Proceedings of the 5th annual symposium on Computer architecture*. ACM Press New York, NY, USA, 1978, pp. 210–215.

[93] A. Davis and S. Nowick, "An introduction to asynchronous circuit design," Unversity of Utah, Tech. Rep., Sep 1997.

[94] M. Dean, T. Williams, and D. Dill, "Efficient Self-Timing with Level-Encoding 2-phase Dual-Rail," in *In Proceedings of the 1991 University of California Santa Cruz Conference*, 1991.

[95] J. Dennis, "First Version of a Data Flow Procedure Language," in *Proceedings of the Programming Symposium*, G. Goos and J. Hartmanis, Eds. Springer-Verlag, 1974.

[96] J. Dennis and D. Misunas, "A Preliminary Architecture for a Basic Dataflow Processor," MIT, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA-02139, Tech. Rep., 1974.

[97] S. Devadas, H. Ma, and A. Newton, "On the Verfication of Sequential Machines at Differing Levels of Abstraction," in *Proceedings of Design Automation Conference (DAC)*, June 1987, pp. 271–276.

[98] D. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1989.

[99] R. Dobkin, R. Ginosar, and C. Sotiriou, "Data Synchronization Issues in GALS SoCs," in *Proceedings of Asynchronous Circuits and Systems (ASYNC)*, 2004.

[100] F. Doucet, S. Shukla, and R. Gupta, "An Environment for Dynamic Component Composition for Efficient Co-Design," in *Proceedings of Design Automation and Test in Europe (DATE)*, 2002.

[101] M. Dwyer, "Automated Analysis of Software Frameworks," in *Workshop on Foundation of Component-Based Systems*, September 1997.

[102] M. Dwyer, G. Avrunin, and Corbett, "Patterns in Property Specifications for Finite-state Verification," May 1999.

[103] J. Ebergen, "A Formal Approach to Designing Delay-Insensitive Circuits," *Distributed Computing*, vol. 5, no. 3, pp. 107–119, 1991.

[104] S. Edwards, "Design Languages for Embedded Systems," Columbia University, Tech. Rep. CUCS-009-03, 2003.

[105] M. Flynn, P. Hung, and K. Rudd, "Deep-Submicron Microprocessor Design Issues," *IEEE Micro*, vol. 19, pp. 11–13, 1999.

[106] E. Friedman, *Clock Distribution Networks in VLSI Circuits and Systems*. IEEE Press, 1995.

[107] ——, "Clock Distribution Networks in Synchronous Digital Integrated Circuits," *Proceeding of the IEEE*, vol. 89, no. 5, pp. 665–692, 2001.

[108] S. Furber, P. Day, J. Garside, N. Paver, and J. Woods, "A Micropipelined ARM," in *Proceedings of VLSI*, 1993.

[109] S. Furber and P. Woods, "Four-Phase Micropipeline Latch Control Circuits," *IEEE Transactions on VLSI Systems*, vol. 4, no. 2, pp. 247–253, 1996.

[110] H. Gageldonk, D. Baumann, K. Berkel, D. Gloor, A. Peeters, and G. Stegmann, "An Asynchronous Low Power 80c51 Microcontroller," in *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, no. 96-107, 1998.

[111] D. Gebhardt and K. Stevens, "Elastic Flow in an Application Specific Network-on-Chip," in *Proceedings of Third International Workshop on Formal Methods on Globally Asynchronous Locally Synchronous Systems (FMGALS)*, 2007.

[112] M. Geilen and T. Basten, "Requirements on the Execution of Kahn Process Networks," in *Proceedings of the 12th European Symposium on Programming (ESOP)*, 2003. [Online]. Available: citeseer.ist.psu.edu/geilen03requirements.html

[113] A. Gerstlauer, *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[114] R. Ginosar, "Synchronization," Tutorial at Fourth ACM-IEEE International Conference on Formal Methods and Models for Codesign, July 2006.

[115] R. Ginosar and R. Kol, "Adaptive Synchronization," in *Proceedings of IEEE International Conference on Computer Design (ICCD)*, 1998.

[116] R. Ginosar, "Fourteen Ways to Fool Your Synchronizer," in *Proceedings of Asynchronous Circuits and Systems (ASYNC)*, 2003.

[117] P. Glaskowski, "Pentium 4 (partially) previewed," *Microprocessor Report*, vol. 14, no. 8, pp. 10–13, 2000.

[118] M. Gordon, *Specification and Verification of Hardware*. University of Cambridge, October 1992.

[119] M. Gowan, L. Biro, and D. Jackson, "Power considerations in the design of the alpha 21264 microprocessor," in *Proceedings of Design Automation Conference (DAC)*, 1998, pp. 726–731.

[120] F. Gruian, P. Roop, Z. Salcic, and I. Radojevic, "The SystemJ Approach to System-Level Design," in *Proceedings of Formal Methods and Models for Co-Design (MEMOCODE)*, 2006.

[121] P. Guernic, M. Borgue, T. Gauthier, and C. Marie, “Programming real time applications with signal,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1335, 1991.

[122] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann, “Polychrony for System Design,” *Journal of Circuits, Systems, and Computers - Special Issue: Application Specific Hardware Design*, vol. 12, no. 3, pp. 261–303, Dec. 2003.

[123] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The Synchronous Data-Flow Programming Language LUSTRE,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[124] N. Hand, “Eliminating Functional Problems due to Clock-Domain Crossings (CDC),” WebCast Presentation from Mentor Graphics and TechOnline Webcasts, May 2006.

[125] ——, “The Need for Automated Clock Domain Crossing Verification Solution,” White Paper: Mentor Graphics, 2006.

[126] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist, “Network on Chip: An Architecture for Billion Transistor Era,” in *IEEE NorChip Conference*, 2000.

[127] G. Hinton, M. Upton, D. Sager, D. Boggs, D. Carmean, P. Roussel, T. Chappell, T. Fletcher, M. Milshtein, M. Sprague, S. Samaan, and R. Murray, “A 0.18-μm CMOS IA-32 Processor With a 4-GHz Interger Execution Unit,” *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1617–1627, 2001.

[128] C. Hoare, “Communicating Sequential Processes,” in *Communications of the ACM*, ser. 8, vol. 21, Aug 1978, pp. 666–677.

[129] ——, *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[130] G. Holzmann, *The SPIN Model Checker*. Addison Wesley, 2004.

[131] P. Hudak and S. Anderson, “Pomset Interpretations of Parallel Functional Programs,” in *Proceedings of the Functional Programming Languages and Computer Architecture*. London, UK: Springer-Verlag, 1987, pp. 234–256.

[132] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley and Sons, 1979.

[133] A. Iyer and D. Marculescu, “Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors,” in *29th International Symposium on Computer Architecture*, 2002, pp. 158 – 168.

[134] A. Jantsch, “Network on Chip,” in *Conference Radio vetenkap och Kommunication*, Stockholm, 2002.

[135] ——, *Modeling Embedded Systems and SOC's Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers, 2003.

[136] ——, "Networks on Chip," Invited seminar at Linkö*ping University, November 2004.

[137] A. Jantsch and H. Tenhunen, *Networks on Chip*. Kluwer Academic Publishers, 2003.

[138] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proceedings of Information Processing*, 1974, pp. 471–475.

[139] J. Kessels, A. Peeters, P. Wielage, and S. Kim, "Clock Synchronization through Handshake Signaling," in *Proceedings of Asynchronous Circuits and Systems (ASYNC)*, 2002.

[140] D. Kim, M. Kim, and G. Sobelman, "Asynchronous FIFO Interfaces for GALS On-Chip Switched Networks," in *International SoC Design Conference*, 2005.

[141] R. Kol and R. Ginosar, "A doubly-latched asynchronous pipeline," in *Proceedings of International Conference on Computer Design (ICCD)*, 1996.

[142] G. Konstadinidis, K. Normoyle, S. Wong, S. Bhutani, H. Stuimer, T. Johnson, A. Smith, D. Cheung, F. Romano, S. Yu, S.-H. Oh, V. Melamed, S. Narayanan, D. Bunsey, C. Khieu, K. Wu, R. Schmitt, A. Dumlao, M. Sutera, J. Chau, K. Lin, and W. Coates, "Implementation of a Third-Generation 1.1-GHz 64-bit Microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1461–1469, 2002.

[143] H. Kopetz, *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[144] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[145] G. Kornaros, D. Pnevmatikatos, P. Vatsolaki, G. Kalokerinos, C. Xanthak, D. Mavroidis, D. Serpanos, and M. Katevenis, "ATLAS I: implementing a single-chip ATM switch with backpressure," *IEEE Micro*, vol. 19, no. 1, pp. 30–41, 1999.

[146] A. Kowalczyk, V. Adler, C. Amir, F. Chiu, C. Chng, W. Lange, Y. Ge, S. Ghosh, T. Hoang, B. Huang, S. Kant, Y. Kao, C. Khieu, S. Kumar, L. Lee, A. Liebermensch, X. Liu, N. Malur, A. Martin, H. Ngo, S.-H. Oh, I. Orginos, L. Shih, B. Sur, M. Tremblay, A. Tzeng, D. Vo, S. Zambare, and J. Zong, "The First MAJC Microprocessor: A Dual CPU System-on-a-Chip," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1609–1916, 2001.

[147] T. Kropf, *Introduction to Formal Hardware Verification*. Springer, 1999.

[148] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary, "Synchronous Elastic Flow," in *Proceedings of Formal Methods on Computer-Aided Design (FMCAD)*, 2006.

[149] N. Kurd, J. Barkatullah, R. Dizon, T. Fletcher, and P. Madland, "Multi-GHz Clocking Scheme for Intel(R) Pentium(R) 4 Microprocessor," in *Proceedings of the 2001 International Solid-State Conference*, 2001.

[150] W. Lam, *Hardware Design Verification*. Prentice Hall, 2005.

[151] E. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 17, no. 12, pp. 1217–1229, 1998.

[152] E. A. Lee and A. L. Sangiovanni-Vincentelli, "Comparing Models of Computation," in *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 1996, pp. 234 – 241.

[153] C.-H. Li, R. Collins, S. Sonalkar, and L. Carloni, "Design, Implementation, and Validation of a New Class of Interface Circuits for Latency-Insensitive Design," in *Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, May 2007.

[154] D. Linder and J. Harden, "Phased logic: Supporting the synchronous design paradigm with delay-insensitive circuitry," *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 1031–1044, 1996.

[155] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," MITLCS, Technical Memorandum TM-373, Nov 1988, tM-351 revised.

[156] N. Lynch and E. Stark, "A Proof of the Kahn Principle for Input/Output Automata," *Information and Computation*, vol. 82, no. 1, pp. 81–92, 1989.

[157] N. Maheshwari and S. S. Sapatnekar, *Timing Analysis and Optimization of Sequential Circuits*. Kluwer, 1999.

[158] M. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Francheschinis, *Modelling with Generalized Stochasic Petri Nets*. John Wiley and Sons, 1995.

[159] A. Marshall, B. Coates, and P. Siegel, "Designing an Asynchronous Communication Chip," *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 8–21, 1994.

[160] A. Martin, "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits," in *Developments in Concurrency and Communication*. Addison-Wesley, 1990.

[161] ——, "The Limitation to Delay-Insensitivity in Asynchronous Circuits," in *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, 1990.

[162] ——, "Asynchronous Datapaths and the Design of an Asynchronous Adder," *Formal Methods on System Design*, vol. 1, no. 1, pp. 119–137, 1992.

[163] D. Mathaikutty and S. Shukla, "SoC Design Space Exploration through Automated IP Selection from SystemC IP Library," in *IEEE International System On Chip Conference*, 2006.

[164] D. Matzke, “Will physical scalability sabotage performance gains?” *IEEE Computer*, vol. 30, no. 9, pp. 37–39, 1997.

[165] K. McMillan, “Interpolation and sat-based model checking,” *Lecture Notes in Computer Science*, vol. 2725, pp. 1–13, 2003.

[166] ——, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[167] J. Mekie, S. Chakraborty, D. Sharma, and G. Venkataramani, “Interface Design for Rationally Clocked GALS Systems,” in *Proceedings of IEEE International Symposium on Asynchronous Circuits and Systems*, 2006.

[168] D. Messerschmitt, *Synchronization Design for Digital Systems*, T. H. Meng, Ed. Kluwer Academic Publishers, 1991.

[169] J.-J. Meyer and E. Vink, “Pomset Semantics for True Concurrency with Synchronization and Recursion (Extended Abstract),” in *Proceedings of Mathematical Foundations of Computer Science (MFCS)*. London, UK: Springer-Verlag, 1989, pp. 360–369.

[170] R. Miller, *Switching Theory Volume II: Sequential Circuits and Machines*. John Wiley and Sons, 1965.

[171] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML - Revised*. MIT Press, 1997.

[172] C. Molnar, I. Jones, W. Coates, J. Lexau, S. Fairbanks, and I. Sutherland, “Two FIFO Ring Performance Experiments,” *Proceedings of the IEEE*, vol. 87, no. 2, pp. 297–307, 1999.

[173] S. Moore, G. Taylor, R. Mullins, and P. Robinson, “Point to Point GALS Interconnect,” in *Proceedings of ASYNC*, 2002.

[174] M. Mousavi, P. L. Guernic, J.-P. Talpin, S. Shukla, and T. Basten, “Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks,” in *Proceedings of Design Automation and Test in Europe (DATE)*, 2004.

[175] R. Mullins and S. Moore, “Demystifying data-driven and pausible clocking schemes,” in *18th UK Asynchronous Forum*, 2006.

[176] T. Murata, “Petri nets: properties, analysis, and applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[177] J. Mutterbach, T. Villiger, and W. Fichtner, “Practical Design of Globally Asynchronous Locally Synchronous Systems,” in *Proceedings of Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2000.

[178] L. Neilson and J. Sparso, “Designing asynchronous circuits for low power: An IFIR filter bank for a digital hearing aid,” *Proceedings of the IEEE*, vol. 87, no. 2, pp. 268–281, 1999.

[179] L. Nielson and J. Spars$\oslash$, "An 85 μW asynchronous filter-bank for a digital hearing aid," in *Proceedings of International Solid State Circuits Conference*, 1998.

[180] S. Nowick, K. Yun, and P. Beeril, "Speculative completion for the design of high-performance asynchronous dynamic adders," in *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997.

[181] S. Oetiker, F. G$\ddot{u}$rkaynak, T. Villiger, H. Kaeslin, N. Felber, and W. Fichtner, "Design Flow for a 3-Million Transistor GALS Test Chip," in *ACiD Workshop*, 2003.

[182] T. Ohba, "A Study of Current Multilevel Interconnect Technologies for 90nm Nodes and Beyond," *FUJITSU Sci. Tech. Journal*, vol. 38, no. 1, pp. 13–21, 2002.

[183] H. Patel and S. Shukla, *SystemC Kernel Extensions for Heterogeneous System Modeling*. Kluwer, 2004.

[184] N. Paver, P. Day, C. Farnsworth, D. Jackson, W. Lien, and J. Lui, "A low-power, low-noise configurable self-timed DSP," in *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, no. 32-42, 1998.

[185] A. Pnueli, "The temporal logic of programs," *18th Annual Symposium on Foundations of Computer Science*, pp. 46–57, October 1977.

[186] D. Potop-Butucaru, B. Caillaud, and A. Benveniste, "Application of Concurrency to System Design," in *ACSD*, June 2004, pp. 67– 76.

[187] ——, "Application of Concurrency to System Design," *Formal Methods in System Design*, vol. 28, no. 2, pp. 111–130, 2006.

[188] D. Potop-Butucaru, S. Edwards, and G. Berry, *Compiling Esterel*. Springer, 2007.

[189] V. R. Pratt, "Modelling concurrency with partial orders," *International Journal of Parallel Programming*, vol. 15, no. 1, pp. 33–71, 1986. [Online]. Available: citeseer.ist.psu.edu/pratt86modelling.html

[190] J. P. Queille and J. Sifakis, "Specification and verification of concurrent programs in CESAR," *Proceedings of the 5th Intl. Symp. on Programming*, vol. LNCS 137, pp. 195–220, 1981.

[191] J. Rabaey, *Digital Integrated Circuits*. Prentice Hall, 1995.

[192] J. M. Rabaey, *Digital Integrated Circuits*. Prentice Hall, 1996.

[193] A. Ranjit, P. Ramkumar, and V. Noel, "Firm IP development: methodology and deliverables," in *Proceedings of Custom Integrated Circuits Conference*, 2000.

[194] H. Schoot and H. Ural, *An improvement on partial order model checking with ample sets*, computer science technical report tr-96-11 ed., Computer Science Technical Report TR-96-11, University of Ottawa, Canada, Sept 1996.

[195] C. Seitz, *System Timing*. Addison-Wesley, 1980, ch. 7, pp. 218–262.

[196] J. Shandle and G. Martin, “Making embedded software reusable for socs,” EE Times, March 2002.

[197] Shi-Yu-Huang and K.-T. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, 1998, vol. 12.

[198] S. Shukla, S. Suhaib, D. Mathaikutty, and J.-P. Talpin, *Next Generation Design and Verification Methodologies for Distributed Control Systems*. Springer, 2007, ch. On the Polychronous Approach to Embedded Software Design, pp. 261–274.

[199] S. Shukla, F. Doucet, and R. Gupta, “Structured component compostion frameworks for embedded system design,” in *Ninth International Conference on High Performance Computing*, 2002.

[200] R. Simone, J.-V. Millo, and J. Boucaron, “Latency insensitive design and central repetitive scheduling,” in *Proceedings of the Fourth International Confererence on Formal Methods and Models for Codesign (MEMOCODE)*, 2006.

[201] M. Singh and M. Theobald, “Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures,” in *Proceedings of Design, Automation and Test in Europe (DATE’04)*, 2004.

[202] A. Sjogren and C. Myers, “Interfacing Synchronous and Asynchronous Modules within a High-Speed Pipeline,” *IEEE Transactions on VLSI Systems*, vol. 8, no. 5, pp. 573–583, 2000.

[203] I. Smarandache, T. Gaurier, and P. L. Guernic, “Validation of mixed signal-alpha real-time systems through affine calculus on clock synchronization constraints.” in *World Congress on Formal Methods*, 1999, pp. 1364–1383.

[204] I. Smarandache and P. L. Guernic, “Affine Transformations in SIGNAL and Their Application in the Specification and Validation of Real-Time Systems,” in *ARTS*, 1997, pp. 233–247.

[205] B. Stroustrup, *VHDL*. McGraw-Hill, 1997.

[206] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla, “Validating Families of Latency Insensitive Protocols,” in *Proceedings of High Level Design Validation and Test Workshop (HLDVT)*, 2005.

[207] ——, “Validating Families of Latency Insensitive Protocols,” *IEEE Transactions on Computers*, vol. 55, no. 11, pp. 1391–1401, 2006.

[208] S. Suhaib, D. Mathaikutty, and S. Shukla, “Framework for composing synchronous ips asynchronously,” Virginia Tech, Tech. Rep. 2005-11, 2005.

[209] ——, “System level design methodology for socs using multi-threaded graphs,” in *Proceedings of IEEE International SOC Conference*, 2005.

[210] ——, “Polychronous methodology for system design: A true concurrency approach,” in *Proceedings of High Level Design Validation and Test Workshop (HLDVT)*, 2006.

[211] ——, “A trace based framework for validation of soc designs with gals systems,” in *Proceedings of IEEE International SOC Conference*, 2006.

[212] ——, “A Formal Refinement-based Framework from Kahn Process Network Specification towards GALS Implementation,” Virginia Tech, Technical Report 2007-04, 2007.

[213] ——, “Dataflow Architectures for GALS,” in *Third International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Design*, 2007.

[214] S. Suhaib, D. Mathaikutty, S. Shukla, D. Berner, and J.-P. Talpin, “A Functional Programming Framework for Latency Insensitive Protocol Validation,” *Electr. Notes Theor. Comput. Sci.*, vol. 146, no. 2, pp. 169–188, 2006, formal Methods on Globally Asynchronous Locally Synchronous Systems.

[215] S. Suhaib, D. Mathaikutty, S. Shukla, and J.-P. Talpin, “True concurrency semantics for polychrony,” Virginia Tech, Tech. Rep. 2006-02, 2006.

[216] G. Sutcliffe, “Automatic Theorem Proving,” http://www.cs.miami.edu/ tptp/OverviewOfATP.html.

[217] I. Sutherland, “Micropipelines,” *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[218] ——, “Micropipelines,” *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[219] S. Sutherland, S. Davidmann, and P. Flake, *System Verilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Kluwer, 2004.

[220] S. Tam, D. Limaye, and U. Desai, “Clock generation and distribution for the 130-nm itanium 2 processor with 6-mb on-die l3 cache,” *IEEE Journal of Solid-State Circuits*, vol. 39, no. 4, 2004.

[221] N. Tansalarak and K. Claypool, “Xcompose: An xml-based component composition framework,” in *Third International Workshop on Composition Languages*, 2003.

[222] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Kluwer, 1998.

[223] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing power in high-performance microprocessors," in *Proceedings of ACM/IEEE Design Automation Conference*, 1998.

[224] J. Udding, "A formal model for defining and classifying delay-insensitive circuits and systems," *Distributed Computing*, vol. 1, no. 4, pp. 197–204, 1986.

[225] M. Vachharajani, N. Vachharajani, D. Penry, J. Blome, S. Malik, and D. August, "The liberty simulation environment: A deliberate approach to high-level system modeling," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 3, pp. 211–249, 2006.

[226] V. Varshavsky and V. Markhovsky, "Gllobal synchronization of asynchronous arrays in logical time," in *Proceedings of 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis*, 1997.

[227] T. Verhoeff, "Delay-insensitive codes- an overview," *Distributed Computing*, vol. 3, no. 1, pp. 1–8, 1988.

[228] T. Villiger, H. Kaeslin, F. G*ü*rkaynak, S. Oetiker, and W. Fichtner, "Self-timed ring for globally asynchronous locally synchronous systems," in *Proceedings of Asynchronous Circuits and Systems (ASYNC)*, 2003.

[229] P. Wolper and D. Leroy, "Reliable hashing without collision detection," in *Proceedings of Computer Aided Verification (CAV)*. Elounda, Greece: Springer Verlag, pp. 59–70.

[230] J. You, Y. Xu, H. Han, and K. Stevens, "Performance evaluation of elastic gals interfaces and network fabric," in *Proceedings of the Third International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Systems (FMGALS)*, 2007.

[231] Z. Yu and B. M. Baas, "Performance and power analysis of globally asynchronous locally synchronous multi-processor systems," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2006.

[232] K. Yun and R. Donohue, "Pausible clocking: A first step toward heterogeneous systems," in *Proceedings of International Conference on Computer Design (ICCD)*, 1996.

[233] ——, "Pausible clocking-based heterogeneous systems," *IEEE Transactions on VLSI Systems*, vol. 7, no. 4, pp. 482–488, 1999.

[234] K. Yun, A. Dooply, and J. Arceo, "The design and verification of a high-performance low control overhead asynchronous differential equation solver," in *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997.

Vita

Syed Mohammed Suhaib

Personal Data

Date of Birth	:	October 29, 1980
Marital Status	:	Single
Visa Status	:	F-1

Office Address:

FERMAT Research Lab.
302 Whittemore Hall
Blacksburg, VA 24061
Email: `ssuhaib@vt.edu`
URL: `http://www.fermat.ece.vt.edu/`
Tel: (540) 257-4444

Research Interests

(a) Formal Methods and Modeling

(b) Formal Verification and Model Checking of Software, Hardware and Real-Time Systems

(c) Latency Insensitive Systems

(d) Globally Asynchronous Locally Synchronous (GALS) Designs

(e) Compositionality of Synchronous Languages

(f) Software for Distributed Architectures

Education

(a) Ph.D., Computer Engineering, (August 2007), Virginia Polytechnic Institute and State University.

Ph.D. Dissertation Title: "Formal Approaches to Intellectual Property Composition Across Synchronization Domains"

(b) M.S., Computer Engineering, (May 2004), Virginia Polytechnic Institute and State University.

M.S. Thesis Title: "An Incremental Methodology for Developing Formal Models"

(c) B.S., Computer Engineering, (May 2002), University of Arkansas.

Current Work

(a) Development and Formal Verification of Globally Asynchronous Locally Synchronous Systems
Design, implementation and formal verification of a protocol for GALS designs.

(b) Pomset-based Semantics for Polychronous Designs
Analyzing properties for software designs for sequential code generation of softwares from concurrent specifications.

(c) Formal Verification of Elastic Controllers
Formal verification of elastic controllers against the properties of elastic machines.

(d) Development and Formal Verification of Latency Insensitive Systems
Implementation and formal verification of latency insensitive single clock and multi-clock synchronous systems.

(e) Type extensions to SCR tool
Proposed ways of implementing new type extensions to the existing SCR tool with arrays, queues, and user-defined types.

(f) Agile Methodology for building SCR models
Developed an agile technique for building formal models of safety critical / High Assurance systems in SCR.

(g) Metamodeling based modeling and verification flow for concurrent application
Developed a metamodeling framework for modeling Multi-Threaded Graph Models and a verification flow to automatically construct timed automata models for UPPAAL model checker.

(h) Extreme Formal Modeling and Verification
Developed an agile formal method named **XFM** based on extreme programming concepts to construct abstract models from a natural language specification of real world complex systems using SPIN and SMV model checkers.

(i) Formal specification and Verification of Real-Time Systems
Modeled and Verified utility accrual Real-Time systems using the UPPAAL model checker.

Skills

– Formal Verification and CAD Tools: NuSMV, Cadence SMV, SPIN, FPV (Intel), IDV (Intel), ModelSim, SCR (NRL), Ptolemy II, Polychrony, Timestool, UPPAAL, MTG, GME.

– Languages: C, C++, SystemVerilog, ForSpec, SystemC, Java, TCL/TK, Standard ML, 8086 Assembly, Visual Basic, SQL, XML.

– Environments: Microsoft Windows, Linux, Solaris, QNX.

Experience

(a) January 2003 to June 2004; September 2004 - May 2005; September 2005 - May 2005; December 2006 - Present: Graduate Research Assistant, Virginia Tech., Blacksburg, VA.

- Developed methodological approach for creating correct GALS designs.
- Designed latency insensitive protocols and developed verification and validation frameworks for latency insensitive designs.
- Analyzed properties of software/hardware designs for deployment over distributed architectures.
- Developed metamodeling based framework for developing multi-threaded graph models and verification flow to UPPAAL model checker.
- Developed extreme modeling based approach to build a model based on the properties and incrementally verifying them simultaneously.
- Applied model extraction schemes for formalizing and verifying systems.
- Used model checking tools for verification of complex systems such as RISC CPU Pipeline, ISA Bus protocol, Smart Home Control System as well as for Utility based protocols for Real-Time scheduling.

(b) May 2006 to November 2006: Student Intern, Strategic CAD Lab (SCL), Intel Corporation, Hillsboro, OR.

- Modeled and verified parametric models of elastic buffers with fork and join structures.
- Formal verification for data consistency of elastic machine with its synchronous counterpart in a validation framework.
- Formal verification of an elastic variable latency model.
- Formal verification of communication protocols (two-way handshake, four-way handshake) for multiclock domains.
- Delivered a library of parametric System Verilog modules and functions for the structural hardware primitives similar to the primitives of the Merlin language.

(c) July 2004 to August 2004; June 2005 - August 2005: Naval Research Lab (Contractor: ITT Industries and Smartronix), Washington, D.C.

- Type extensions to SCR tool.
- Translator for type extended SCR to currently acceptable SCR.
- Agile methodology for developing SCR models of safety critical / High Assurance Systems.
- Incremental development of PACS system using SCR.
- Techniques for bridging gap between the designer and customer.

(d) August 2002 to December 2002: Technical Support for Law School Computing Services, University of Arkansas, AR.

- Provided Support for Network Administrator including configuring of systems on the network, maintaining and upgrading systems and maintaining School of Law web-page.

Journals and Book Chapter

S. Shukla, S. Suhaib, D. Mathaikutty, and J.-P. Talpin. On the Polychronous Approach to Embedded Software Design, pages 261-274. In *In Next Generation Design and Verification Methodologies for Distributed Control Systems.* Springer, 2007.

S. Suhaib, D. Mathaikutty, D. Berner and S. Shukla. Validating Families of Latency Insensitive Protocols. In *IEEE Trans. Computers 55(11): 1391-1401 (2006).*

S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. XFM : An incremental methodology for developing formal models. In *ACM Trans. Design Autom. Electr. Syst. 10(4): 589-609 (2005).*

P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi. A Formally Verified Application-Level Framework for Utility Accrual Real-Time Scheduling On POSIX Real-Time Operating Systems. In *IEEE Trans. Software Eng. 30(9): 613-629 (2004)*.

D. Berner, S. Suhaib, S. Shukla, and J. Talpin. Capturing Formal Specification into Abstract Models, pages 325–346. In *Formal Methods and Models for System Design.* Kluwer Academic Publishers, 2004.

Conference Proceedings and Workshops

S. Suhaib, D. Mathaikutty, and S. Shukla. Dataflow Architectures for Globally Asynchronous Locally Synchronous Designs. In *3nd International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS'05),* July 2005.

S. Suhaib, D. Mathaikutty, S. Shukla, and J.-P. Talpin. Polychronous Methodology for System Design: A True Concurrency Approach. In *IEEE International High Level Design Validation and Test Workshop (HLDVT'06),* November 2006.

S. Suhaib, D. Mathaikutty and S. Shukla. A Trace Based Framework for Validation of SoC Designs with GALS Systems. In *IEEE International SOC Conference (SOCC' 06),* September 2006.

S. Suhaib, D. Mathaikutty, D. Berner, J. P. Talpin and S. Shukla. A Functional Programming Framework for Latency Insensitive Protocol Validation. In *2nd International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS'05),* July 2005. (Also in) Electr. Notes Theor. Comput. Sci. 146(2): 169-188 (2006)

S. Suhaib, D. Mathaikutty, D. Berner and S. Shukla. Validating Families of Latency Insensitive Protocols. In *IEEE International High Level Design Validation and Test Workshop (HLDVT'05),* November 2005.

S. Suhaib, D. Mathaikutty, and S. Shukla. System Level Design Methodology for System On Chips using Multi-Threaded Graphs. In *IEEE International SOC Conference (SOCC' 05),* September 2005.

S. Suhaib, D. Mathaikutty, and S. Shukla. Effects of property ordering in an incremental formal modeling methodology. In *IEEE International High Level Design Validation and Test Workshop (HLDVT'04)*, November 2004.

S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Extreme formal modeling for hardware models. In *5th International Workshop on Microprocessor Test and Verification (MTV'04),* September 2004.

S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Property ordering effects in an incremental formal modeling methodology. In *Thirteenth International Workshop on Logic and Synthesis (IWLS'04),* June 2004.

D. Bhaduri, M. Chandra, H. D. Patel, S. Sharad, and S. Suhaib. Systematic abstraction of microprocessor rtl models to enhance simulation efficiency. In *4th International Workshop on Microprocessor Test and Verification (MTV'2003)*, June 2003.