Latency Insensitive Design for Multiple Clock SoC

A thesis submitted in partial fulfillment of
requirements for the third semester of degree of

Master of Technology

in

Computer Science & Engineering

Submitted by

Shirshendu Das

08410112

Under the guidance of

Dr. Hemangee K. Kapoor

Department of Computer Science and Engineering

Indian Institute of Technology, Guwahati

Abstract

Latency insensitive (LI) design has been proposed as the correct by construction methodology for coping with latency arising due to long interconnects in a single clock SoC. GALS systems are also a robust framework for solving several issues in the design of complex hardware systems. Thus, in order to build a general system, we need both the approaches from LI design and GALS architecture. Our work builds the mathematical foundations for such type of systems.

We extend the single clock LI theory to handle latency issues in connecting multiple clocked modules. Illustrations are given to show why the single clock LI design cannot be applied to this new domain. The theory given can be used to connect modules with rationally related clock frequencies and also for frequencies having phase differences.

A multiple clock patient process is defined and actions needed to handle stalls as well as back-pressure are provided. A definition for an appropriate relay station is given, which can be plugged along with the relay stations of the single clock domain. The new relay station is thus acting as a converter/link between the two timing domains. Proofs for composability of such systems are given and the limiting case of a single clock relay station is derived from the multiple clock relay station.

Contents

List of Figures

Chapter 1

Introduction

As the commercial demand of System-on-Chip (SoC) based products are increasing it become very necessary to design the chips in short time and also with a minimum cost. For achieving these two constraints the designer should consider the followings:

- The design process of the SoC must be very fast. They have to choose some design procedure so the design cycle need not repeat multiple times.
- They have to reuse the existing intellectual-property modules (known as IP cores) to design the SoC's.

But the main problem in designing a SoC in quick time is to handle the communication delays between the different modules of SoC. Even the number of layers and aspect ratios are increasing; the resistance-capacitance delay of an average metal line having a constant length is getting worse with each process generation [**?**, **?**]. The operating frequency, die size and average interconnect length are also increasing. Combination of all these effects making the interconnect delay larger than a clock cycle. According to [**?**] a signal requires more than ten clock cycles to traverse the entire chip area. Also [**?**] estimated that in one clock cycle a signal can travel only 0.4% to 1.4% of the total chip area. Hence the key property to increase the performance of the overall design is to minimize the on-chip distance traveled by a signal.
It is believed that the recent CAD tools containing logic synthesis and physical design will suffer from the impracticality of accuracy estimating the latency of global wires [**?**]. A small variation in the input specification (hardware description language) can

lead to major variations in the final design. The main assumption in synchronous design is that the delay of each combinational path is less than one clock cycle [**?**]. Combinational path means the path traveled by a signal starting from one latch and only going through the combinational logic and wires to reach another latch. The slowest combinational path dictates the maximum operating frequency for the system. Once the final layout have been derived by the designer, every path having delay more than one clock cycle simply represents an exception and must be fixed. To fix these exceptions designers use techniques like wire buffering, transistor resizing to reroute the wires, replacing the modules and if nothing works then they have to redesign the entire system. So the design process repeats multiple times to minimize the on chip delays. But even there is no guarantee to get a minimum *on-chip delay* design in finite time.

Existing IP cores can be used to design new SoC. Using these IP cores the designers can save both the time and cost. IP cores were actually functional blocks designed for previous generation within the same vendor. Now IP cores are also available as a standard functional unit designed by some specialized vendors. An IP core must have the capability to communicate with other modules in different environment. Designing a new SoC requires assembling the existing IP cores with some modifications on them. The main challenges in using these IP cores is the synchronization issues that will arise naturally while assembling predesigned component. Researchers proposed the solution that there should be a orthogonalization of the factors effecting the design in order to increase the complexity of these systems [**?**]. System on Chip design essentially consists of two parts. The first one is the computation performed by each of the IP cores and the second one is the communication between them. It is believed that the design paradigm will now shift from computational bound to communication bound designs. Under these circumstances, the Latency Insensitive design will automatically gain attention as it serves the need of the hour. It uses the existing IP cores to design the new SoC's without going through system rerouting or redesigning. Also, latency is influencing the design stages of state at microprocessors. Few stages of pipelining called the "drive" stages" of Netburst architecture of Pentium 4 are initiated by latency [**?**].

1.1 Latency Insensitive Design

The Latency Insensitive (LI) design actually based on latency insensitive protocol proposed by Carloni et el.[2]. This protocol controls the communication among the different modules of a patient system. Here patient system means a synchronous system whose functionality depends only on the order of each signaling events and not there exact timing. For example consider that there are two signals entering in a system as input and another signal is coming out from the system as output which carrying the result of some functional computation performed with the input signals. Now if the system is patient then for any arbitrary number of delays in the input signals the output signal will always consist of same ordering sequence of events. The timing of events in a sequence may not be similar with another sequence but the ordering of there events will remain same.
Designer can model a synchronous system as a set of modules. These modules are communicating with each other by exchanging signals on a set of point-to-point channels. The protocol guarantees that a system, if composed of functionally correct modules, behaves correctly, independent from the delays in the channels connecting the modules. An hardware implementation of the system is possible to synthesize automatically in such a way that the functional behavior of the system is robust with respect to large variation in communication latency. This methodology orthogonalizes computation and communication because it separates the module design from the communication architecture options, while enabling automatic synthesis of the interface logic. The separation is usefull in two ways:

- Module design becomes simple because designer can assume the synchronous hypothesis. That is the intermodule communication will take no time (or in other words, it's completed within one virtual clock cycle.)
- It's permits the exploration of tradeoffs in deriving the communication architecture up to the design process late stage, because the protocol gurantees that the interface logic can absorb arbitrary latency variations.

A brief description of latency insensitive design flow is given below:

- Collecting some synchronous components. These components can be custome modules or IP cores.

- Encapsulate each component to an automatically generated shell to make them patient. A shell is a collection of buffering queues one for each port and a control logic that interfaces the component with the LI protocol.

- Physical design. In this step the designer design the system by using standard CAD tools. This process considers the delay of each channel as zero time of a virtual clock.

- Inserting relay stations: designers segment every wire whose latency is grater than the clock period of real clock by distributing the necessary relay stations. The key idea is borrowed from pipelining: partition the long wires into segments whose length satisfy the timing requirement imposed by the real clock. We will explain relay station later in this chapter.

The only necessary precondition in this method is that the encapsulated components are stallable, that is they can freeze their operation for an arbitrary time without loosing there internal state.

Let us consider that we want to design a system with four modules m_1, m_2, m_3 and m_4. Consider that m_1 will send information to m_2, m_3 and m_4. Also m_3 needs to send information to m_2 and m_4. The design process start by taking four IP cores one for each module. After this, each IP core must have to encapsulate with an automatically generated shell to make the core patient. Let us called these combinations (IP core encapsulated with shell) as patient modules. Now with these four patient modules the design tool design the system layout without considering the delays in the communicating wires. Figure 1.1(a) is showing a sample layout design. After designing this layout we have to calculate the communication delays of each wires. In our figure these are shown by numbering each wires. Now we can see from the figure that there are two wires having delay more than one clock period. To solve these problems we now need to insert relay stations in the corresponding wires (as shown in fig 1.1(b)) to divide them into segments having delay of one clock period. Now since each modules are patient and even the relay stations are also patient [2] (we will explain later), the entire system must be a patient system [2] (explanation is in next chapter).

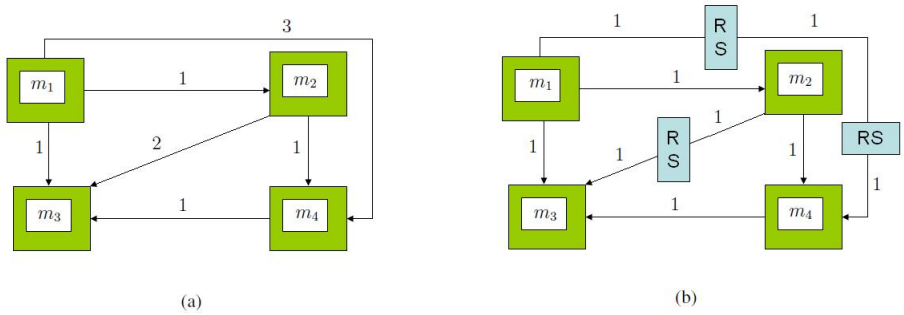

Figure 1.1: *An example of Latency Insensitive(LI) design.*

1.1.1 Channels and back-pressure

Channels are point-to-point unidirectional links between a source and a sink module. Data are transmitted on a channel by means of packets that consist of variable number of fields. We consider only two fields: payload, which contains data; and void, a one bit flag that, if set to 1 denotes that no data are present in the packet. If a packet contains meaningful payload then we call it as true packet.

A channel is a combination of wires and relay stations. There should be always a finite number of relay stations in a channel. These relay stations are actually representing the buffering capability of a channel. At each clock tick the source module will either put a new true packet into the channel or put a void packet on it if there is no true data to send. From now onwards we represent the event of putting a void data into a channel as inserting a delay into the channel. Conversly, the sink module will receive the new incoming packet from the channel. The receiving packet may be true or void.

Every module maintains a separate queue for its each input and output channels. A source might not be ready to send a true packet, and a sink module might not be ready to receive it if, for instance, its input channel is full. Since the latency insensitive protocol demands fully reliable communication among the modules, it requires the proper delivery of all packets. To achieve this, channel definitions are slightly modified to allow a bit of information to move in the opposite direction. Figure 1.2 shows the signals as dotted lines. It is similar to the NACK signal in the request/acknowledge protocols of asynchronous design. This back pressure mechanism controls the flow of

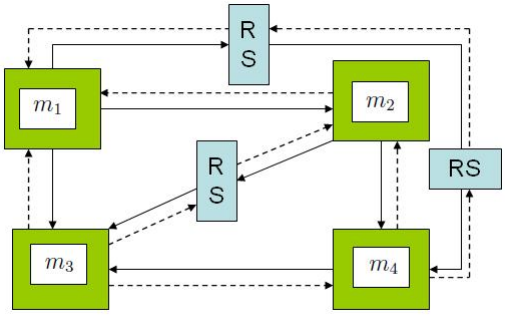

Figure 1.2: *An example showing backpressure in Latency Insensitive design.*

information on a channel while guaranteeing that no packets are lost.

1.1.2 Shell encapsulation

It is possible to devise a method to automatically synthesize an instant of a shell as a wrapper to encapsulate a module. We can also interface the shell with the channels so that the module becomes a patient system. The only necessary pre-condition for doing this is that the module must be stallable. At each clock cycle, the module's internal computation must fire only if all inputs required for that computations have arrived. The module shell's first task is to gurantee this input synchronization. Its second task is to output propagation. At each clock cycle, if the module has produced new output value and no output channenl has previously raised a stop flag, then the shell can transmit these output values by generating new true packets. If the shell does not verify either of these conditions, then it must transmit a void packet. So in summary, a shell for a module cyclically performs the following tasks:

- Filtering away the void packets from the incoming packets and extracting the true packets
- When all he input values required for the next operation are become available, it allows the module to perform the operation.
- It obtains the computation results from the module.

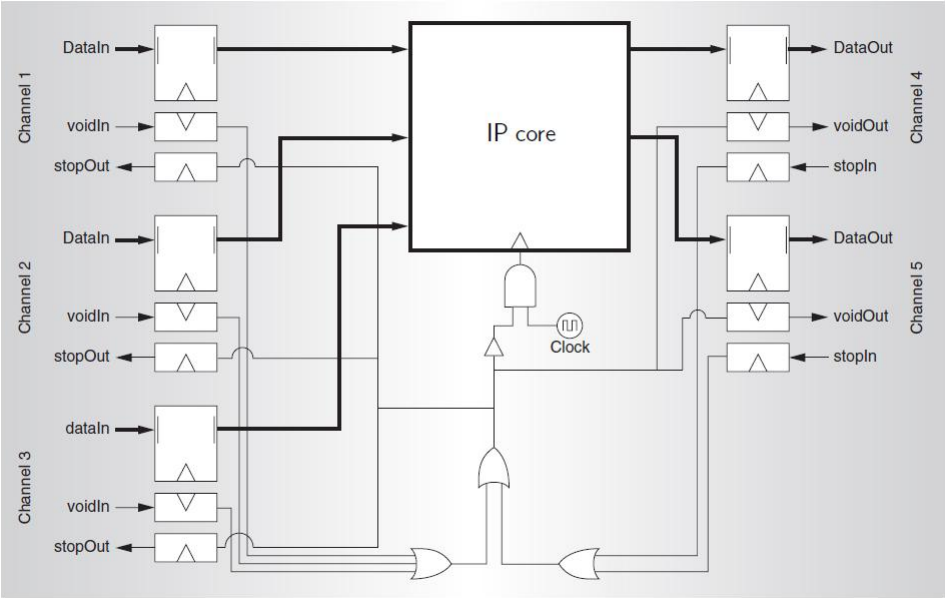

Figure 1.3: *Shell encapsulation: Making an IP core patient. (taken from [?])*

- It routs the result into the output channels if no output channel has previously raised a stop flag; otherwise it stores the result in the queue and sends a void packet. Note that in every clock cycle it will send a packet either true or void.

Figure 1.3 taken from [?] illustrates a simple unoptimized implementation of a shell wrapping a module having three input and two output channels.

1.1.3 Relay stations

As we mentioned before, relay stations are inserted within the wires to divide them into segments. Now if we consider each segment as a separate channel then relay station is a process connecting two channels. In the next chapter we will explain the properties and the proof of the theorem given in [2], which says that relay station is a patient process.

1.2 Motivation

Recent advances in VLSI technology has made it possible for integration of complex systems into SoC design. Higher power integration and deep pipelining has raised concerns regarding the power consumption and clock skew over in a synchronous SoC. This has lead to the new paradigm of Globally Asynchronous Locally Synchronous (GALS) design. This design divides the SoC into various domains called Synchronous Blocks (SBs) and they interact with each other using asynchronous communication protocols. The modules in a block are synchronous with each other and can communicate with each other using either synchronous or asynchronous protocols according to the design requirement. Here we have two paradigms which propose solutions to the design problems today. Attempts have been made by researchers to apply LI Design for GALS systems. In this work, we attempt to provide a denotational semantics for a particular design of multiple clock latency insensitive (MCLI) design and thus prove the correctness of the LI design for multiple clock systems. We call the systems, designed with "MCLI design" as MCLI systems. The existing denotational model of the LI system only provides semantics for single clocked relay stations (i.e. both the sender and the receiver have the same clock frequency with no jitter and skew). Thus we did this work to extends the relay station to support multiple clock domain. Lastly we derive a process algebraic model for our proposed MCLI system. For this we use the process algebraic language "Timed CSP" [**?**], which is an extension of "CSP" [**?**].

1.3 Organization of the report

Our work is mainly divided into two parts:

1. **Proposing a solution for MCLI design and a denotational framework for expressing it.**

2. **Presenting a multiple-clock process-algebraic model for our proposed MCLI system.**

This chapter introduces the basic features and the need of Latency Insensitive (LI) design. Later we examine the current state of art and view at the drawbacks of

the existing LI design methodology and semantics of LI design. We then propose a solution for MCLI design and also a denotational framework for expressing it. After this we build a "Timed CSP" [**?**] model (process algebraic) for our proposed solution of MCLI design. The rest of the chapters are organized as follows: in chapter 2 we provide the literature review and the detail description of the existing denotational framework for expressing Single Clock LI design. We used many notations, definitions and theorems from this existing framework to propose and prove the correctness of our own framework. In chapter **??** we propose our whole work of part1 as defined above. Chapter 4 is dedicated for the literature review of process algebra. There has been many languages (process algebraic) for modeling interactive systems depending on their nature (clock-less, single-clock or multiple-clocks). In this chapter we discuss different existing languages of processes algebra and their pros and cons. Also we explain the existing process-algebraic model for single-clock LI systems proposed in [1]. Our proposed model for MCLI system is actually an extension of this model [1]. In Chapter 5, we propose a process algebraic model for our MCLI system using "Timed CSP". And finally in Chapter 6, we conclude with future works.

Chapter 2

Related Works-1

In this chapter we are going to discuss some important works about both single clock and multi clock LI design. Single clock LI design is now a fully successful chip design procedure. Many authors contributed for single clock LI design after it has been proposed by Carloni et el. [2, **?**]. Multiclock LI design is still in experimental phase and several peoples are working on it. Many useful concepts about multiclok LI designed has been proposed, but no one has given any denotational semantics for expressing and proving its correctness. In the first section of this chapter we discuss some important and interesting works about single clock LI design. Second section discuss about the current status of multiclock LI design. The remaining sections describe the denotational semantics used by Carloni et el. for the theory of single clock LI design [2].

2.1 Single Clock Domain

Latency-insensitive system were originally proposed by Carloni et el. [2, **?**, **?**, **?**] for the design of single clock SOC's. If an IP block is not latency-insensitive then they proposed to make it latency-insensitive by encapsulating the IP by a simple wrapper circuit proposed by them. The main concepts of this work are already discussed in chapter 1. In this method, in order to manage interconnect latency, relay stations need to be inserted along the interconnect. Later many interesting implementations of the LI design protocol came into existence [**?**]. The performance analysis of LI

designed system has been analyzed in [**?**]. But this analysis has not considered the "back-pressure", which is important for fully reliable communication in LI designed systems. The performance analysis done at [**?**] using Max Plus algebra formally proved the performance upper bound, achievable by LI design. They also proved that their proposed implementation of LI protocol provides robust communication through back-pressure. A model of the relay stations and shell wrappers in SyncCharts was described in [**?**]. There has been a Process Algebra model of the LI Design [1]. In this paper the author has given the definitions for LI computational blocks and LI connectors. Some conditions are also given to check the liveness and deadlock freedom of LI systems. The paper was actually a step toward the high-level specification and verification of LI systems.

2.2 Multiple Clock Domain

In order to extend the single clock latency-insensitive design proposed by Carloni et el. [2] to multiclock many authors have done many useful works. In [**?**] and [**?**] the authors propose four new FIFO designs. Each design is for various combinations of synchronous and asynchronous systems. The basic implementation of LI methodology for the GALS architecture is proposed in [**?**]. In [**?**] the authors proposed a method of addressing the combined problem of multiple clock frequencies and implementation-induced latencies and clock skews. In this method different clock frequencies are managed using the rate multipliers proposed by [**?**]. In [**?**] and [**?**], the authors examine systematically the problem of designing interface circuits for rationally clocked components in GALS systems.

2.2.1 GALS design style

In [**?**] the authors categorize GALS design into three different classes:

- plausible clocks.
- asynchronous interface.
- loosely synchronous interfaces

Plausible clock is used to enable separate clock domains to communicate without metastability. Each locally synchronous blocks generates its own clock with a ring oscillator. Each ring oscillator period is set according to the speed requirements of the block it derives. One way of pausing is to pause temporarily [**?**, **?**] or stretch [**?**] the receiver's clock. Such designs require "wrapper logic" at the receiver. Its main drawback is, the receiver's clock has to be started again and again. A design by Myers et el. [**?**] which interfaces asynchronous to synchronous domain uses this approach of pausing the clocks. Pausing delays a clock sampling edge until after the arrival of data from the other domain, thus avoids metastability altogether.

The second GALS design style is the asynchronous interface. This methods uses circuits known as synchronizer to transfer signals arriving from an outside timing domain to the local timing domain. Although simple asynchronous interfaces suffer from low throughput, this limitation can be overcome with careful designs. Asynchronous interfaces offer the most flexibility and probably the easiest integration into the exiting CAD flows.

The third style, loosely synchronous interface, arises when some bounds on the frequencies of communicating blocks are known. In this style, the designer exploits these bounds to ensure that timing requirements are met. This style requires timing analysis on the paths between the sender and receiver and is less amenable to dynamic changes in the clock frequency. This analysis however, makes handshaking unnecessary during data transfer, so the resulting circuits can archive higher performance and have more deterministic latencies than those of the other methods.

2.2.2 Multi-clock interconnects

There are three major interconnect designs proposed in [**?**], [**?**], [**?**] and [**?**]. We give a brief description of them as follows:

2.2.2.1 Self-Timed Interfaces for Crossing Clock Domains

The full details of this design can be found in [**?**]. The advantage of this design is that it not only deals with multi-clock interconnect, but also handles the jitter and clock delay induced. The basic model is the use of a synchronizer which takes the

input from both the clocks of receiver and sender and designs a new clock pulse that does not have a synchronization problem with either the sender or the receiver. The paper starts with a design where the modules are running on same clock frequency but having jitter and then extends the same design to modules where the clock frequencies are rationally related and unrelated clock cycles. The disadvantage is that the design is complex and the scope of this design is restricted.

2.2.2.2 Interface for rationally clocked GALS systems

The full details of this design can be found in [**?**]. The main idea of this paper is to examine two different aspects of multi-clock interconnect, namely flow control and synchronization. They explained that these two phenomenon are orthogonal but elimination of one does not necessary means the elimination of the second. This paper then proceeds to enplane each of these aspects in detail and thus comes up with a design which is free from both the problems of flow control and synchronization. A method to implement this design for rationally related frequencies is also given. The design is simple and robust is only applicable to a restricted designs.

2.2.2.3 Robust interfaces

The full details of design can be found in [**?**]. This paper is somewhat similar to [**?**] and gives a description of a FIFO which acts as multi-clock interconnect. The FIFO has input at one clock frequency and output at a different clock frequency. The FIFO maintains a *begin* and *end* flag which define the beginning and ending of the FIFO. Two signals empty and full are connected to receiver and sender respectively. It is a general interconnect and can be used to connect domains of unrelated and rationally related clock frequencies. This design is simple and robust, but there is a requirement of use of synchronizer in some cases.

2.3 Existing Theory

As we mentioned previously, theory of single clock LI design was proposed by Carloni et el. [2]. Now we are giving a brief explanation about there proposed theory. This

explanation is necessary because we are using similar framework for our proposed theory (for multiple clock LI design). We used many notations, definitions and theorems from this existing theory to extend it for multiple clocks. Carloni et el. [2] used a previous framework called *Tagged Signal Model*, which has been proposed by Lee and Sangiovanni Vincentelli, to represent complex system as a collections of signals and processes [**?**]. In the remaining two sections of this chapter we will explain both *Tagged Signal Model* as well as the theory proposed by [2].

2.4 Tagged Signal Model

The formal theory behind the LI design is based upon a *tagged signal model* [**?**]. which can be used to represent complex systems as a collections of signals and processes. Below we are explaining some concepts of *Tagged Signal Model* which are useful for both existing and our proposed theory.

The basic element of the *tagged signal model* is the object called *event*. An event has a tag and a value. Tag is used to model time, precedence relationship and synchronization point etc. where the value is used to represent the operands and results of computation. If $\mathcal{V}$ represents the set of values and $\mathcal{T}$ represents the set of tags then an *event* e is defined as a member of $\mathcal{V} \times \mathcal{T}$.

A signal s is defined as $s \subset (\mathcal{V} \times \mathcal{T})$. It can also defined as a set of events. If two events have the same tag then the events are called *synchronous* events. Similarly two signals s_1 and s_2 are said to be *synchronous* if their corresponding events have same tag. So all synchronous signals contains same set of tags.
The set of all signals is denoted by S where S is a power set of $\mathcal{V} \times \mathcal{T}$. An *N tuple* of signals is defined as the set of *N* signals. The set of all such N tuples is denoted by $\mathcal{S}^N$. A process $\mathcal{P}$ is a subset of $\mathcal{S}^N$. A particular *N tuple* $nt \in \mathcal{S}^N$ is said to satisfy a process P if $nt \in P$. The *N tuple*, which satisfies the process is called a *behavior* of the process. So the process can also be defined as a set of behaviors. If $\{P_1, \ldots, P_M\}$ be a set of processes of same sort [**?**] then the *composition* of these processes is a new process P, defined as $P = \bigcap_{i=1}^{M} P_i$.

If $J = (j_1, \ldots, j_m)$ be an ordered set of indexes in the range $1 \leq j \leq N$ then the $projection$ $\pi_I(b)$ of a behavior $b = (s_1, \ldots, s_N) \in \mathcal{S}^N$ onto $\mathcal{S}^m$ is defined as $\pi_J(b) = (s_{j_1}, \ldots, s_{j_m})$. Given a process $P \subset \mathcal{S}^N$, the projection $\pi_I(P)$ can be defined as the set $(s' | \exists s \in P \wedge \pi_J(s) = s')$. A simple process where two (or more) of the signals are constrained to be identical is called a $connection$ and is denoted as C. For example $C(i,j) \subset \mathcal{S}^N : (s_1, \ldots, s_N) \in C(i,j) \Leftrightarrow s_i = s_j$, with $i, j \in [1, N]$.

Inputs are events or signals that are defined outside the process. Many processes have the notion of inputs. More formally an *input* to a process $P \subset \mathcal{S}^N$ is an externally imposed constraints $A \subset \mathcal{S}^N$ such that $A \cap P$ is the set of total acceptable behaviors. The set of all possible inputs $B \subset \mathcal{S}^N$ is a further characterization of a process. If P be a process and b represents its inputs then P is said to be *closed* if $B = \{\mathcal{S}^N\}$, a set with only one element $A = \mathcal{S}^N$. Since the set of behaviors is $A \cap P = P$, there are no input constraints in a closed process. The set of signals of a process P can be partitioned into three disjoint subsets by partitioning the index set as $\{1, \ldots, N\} = I \cup O \cup R$, where I is the ordered set of indexes for the input signals of P, O is the ordered set of indexes for output signals of P and R is the ordered set of indexes of the remaining signals of P. A process is $functional$ with respect to (I, O) if for every behaviors $b \in P$ and $b' \in P$ where $\pi_I(b) = \pi_I(b')$, it follows that $\pi_O(b) = \pi_O(b')$. Hence we can completely characterize a functional process P by the tuple (F, I, O) where F is a function a function $F : \mathcal{S}^{|I|} \rightarrow \mathcal{S}^{|O|}$. A process is determinate if for any input $I \in B$, it must have exactly one behavior or exactly no behaviors. In other cases the process is considered as *nondeterminate*.

If a system (set of processes) is synchronous then all the signals belonging to this system will be synchronous with each other. A totally ordered set *timestamps* of tags $\mathcal{T}$ is used in timed systems. Given a signal s, the natural ordering of its events is represented by the *timestamps* ordering of s. A functional process is (strictly) causal if two outputs can only differ at the timestamps that (strictly) follow the timestamps when the inputs producing these outputs show a difference. A functional process P is said to be causal if $\forall s_i, s_j \in \mathcal{S}^{|I|}(d(F(s_i), F(s_j)) \leq d(s_i, s_j))$. Similarly P is strictly causal if $\forall s_i, s_j \in \mathcal{S}^{|I|}(d(F(s_i), F(s_j)) < d(s_i, s_j))$. Here d represents a metric on the set $\mathcal{S}^N$ of N-tuples of signals.

2.5 Theory of LI design

2.5.1 Important definitions about LI design

Definition 2.5.1. *Function* $\sigma : \mathcal{S} \times \mathcal{T}^2 \rightarrow \sum_{lat}$ *represents the sequence of values of a signal. This function takes a signal* $s = \{(v_0, t_0), (v_1, t_1), \ldots\}$ *and an ordered pair of timestamps* $(t_i, t_j), i \leq j$ *and returns a sequence* $\sigma_{[t_i,t_j]} \in \Sigma_{lat}$ *s.t.* $\sigma_{[t_i,t_j]}(s) = v_i, v_{i+1}, \ldots, v_j$.

Example 2.1. Consider the signal s_2 of behavior b in the figure 2.1. We have

$$\sigma_{[t_0,t_6]}(s_3) = i_1, i_2, \tau, \tau, i_3, \tau, i_4$$

Definition 2.5.2. *The function* $\mathcal{F}_\iota : \Sigma_{lat} \rightarrow \Sigma^*$ *returns only the informative events from a given sequence of above function. Mathematically the function returns a sequence* $\sigma' = \mathcal{F}_\iota[\sigma]$ *s.t*

$$\sigma'_i = \begin{cases} \sigma_{[t_i,t_i]}(s), & \text{if } \sigma_{[t_i,t_i]}(s) \in \Sigma \\ \epsilon, & otherwise \end{cases}$$

Example 2.2. If we take the signal s_2 of behavior b from the figure 2.1, then we have

$$\mathcal{F}_\iota[\sigma_{[t_0,t_6]}(s_3)] = i_1, i_2, i_3, i_4$$

Definition 2.5.3. *The function* $\mathcal{F}_\tau : \Sigma_{lat} \rightarrow \{\tau\}^*$ *returns only the stalling events from a given sequence of above function* σ. *Mathematically the function returns a sequence* $\sigma' = \mathcal{F}_\tau[\sigma]$ *s.t*

$$\sigma'_i = \begin{cases} \sigma_{[t_i,t_i]}(s), & \text{if } \sigma_{[t_i,t_i]}(s) \in \tau \\ \epsilon, & otherwise \end{cases}$$

Example 2.3. If we take the signal s_2 of behavior b from the figure 2.1, then we have

$$\mathcal{F}_\tau[\sigma_{[t_0,t_6]}(s_3)] = \tau, \tau, \tau$$

$$b = \begin{cases} s_1 = & i_1, i_2, i_3, i_4, i_5, i_6, i_7, \tau, \ \tau, \ \tau, \ \tau, \tau, \ldots \\ s_2 = & i_1, i_2, \tau, i_3, i_4, i_5, \ \tau, \ i_6, i_7, \tau, \ \tau, \tau, \ldots \\ s_3 = & i_1, i_2, \tau, \ \tau, \ i_3, \ \tau, \ i_4, \ i_5, \ \tau, \ i_6, \tau, i_7 \ldots \end{cases}$$

Figure 2.1: *An example of a behavior with three signals.*

Definition 2.5.4. *Given a signal s is said to be strict if and only if all informative events precede all stalling events. The signals which are not strict are called delayed signal.*

Example 2.4. In figure 2.1, the signal s_1 of behavior b is a strict signal where the remaining two signals are delayed signals.

Definition 2.5.5. *Two signals are said to be latency equivalent if they have the same sequence of informative events. It means that the two signals are identical except for different delays between two successive informative events. Mathematically two signals s_i and s_j are latency equivalent $s_i \equiv_\tau s_j$ iff $\mathcal{F}_\iota[\sigma(s_i)] = \mathcal{F}_\iota[\sigma(s_j)]$.*

Example 2.5. In figure 2.1, all three signals of behavior b are latency equivalent with each other.

Definition 2.5.6. *Given a class of latency equivalent signals a strict signal which is latency equivalent with them is called a reference signal of that class of signals.*

Example 2.6. In figure 2.1, the signal s_1 is the reference signal of the remaining two signals of behavior b.

Definition 2.5.7. *The ordinal of an informative event coincides with its position in the reference signal. Mathematically the ordinal of an informative event $e_k = (v_k, t_k) \in \mathcal{E}_\iota(s)$ is defined as $ord(e_k) = \mathcal{F}_\iota[\sigma_{[t_0,t_k]}](s) - 1$.*

Example 2.7. In figure 2.1 $ord(e_0(s_2) = 0), ord(e_4(s_2)) = 3$ and $ord(e_4(s_3)) = 2$.

Definition 2.5.8. *If $b_1 = \{s_1, \ldots, s_N\}$ and $b_2 = \{s'_1, \ldots, s'_N\}$ two behaviors then they are said to be latency equivalent iff $\forall i, (s_i \equiv_\tau s'_i)$. If all the signals in a behavior are strict signals then the behavior is called strict behavior. Every class of latency equivalent behaviors contains only one strict behavior. This is called the reference behavior.*

Definition 2.5.9. *Two processes P and P' are said to be latency equivalent $P \equiv_\tau P'$, if every behavior of P is latency equivalent to some behavior of P'. A process is said to be a strict process iff every behavior b is strict. Every class of latency equivalent processes contains only one strict process and it is called the reference process.*

Definition 2.5.10. *Given a behavior $b = (s_1, , s_N)$, symbol $\leq_c$ denotes a well-formed order on its set of signals. The well-founded order induces a lexicographic order $\leq_{lo}$ over the set of informative events of b, s.t. for all pairs of events (e_1, e_2) with $e_1 \in \mathcal{E}_\iota(s_i)$ and $e_2 \in \mathcal{E}_\iota(s_j)$*

$$e_1 \leq_{lo} e_2 \Leftrightarrow [(ord(e_1) < ord(e_2)) \vee ((ord(e_1) = ord(e_2)) \wedge (s_i \leq_c s_j))] \tag{2.1}$$

Example 2.8. If we consider that the three signals of behavior b in the figure 2.1 have the relationship $s_1 \leq_c s_2 \leq_c s_3$ then the ordering of the informative events in this behavior will be:

$$e_0(s_1), e_0(s_2), e_0(s_3), e_1(s_1), e_1(s_2), e_1(s_3), e_2(s_1), e_3(s_2), e_4(s_3), e_3(s_1), e_4(s_2), \ldots$$

Definition 2.5.11. *Given a behavior $b = (s_1, , s_N)$ and an informative event $e(s_i) \in \mathcal{E}_\iota(s_i)$, the function nextEvent is defined as*

$$nextEvent(s_j, e(s_i)) = \underbrace{min}_{e_k(s_j) \in \mathcal{E}_\iota(s_j)} \{e(s_i) \leq_{lo} e_k(s_j)\} \tag{2.2}$$

Example 2.9. In the figure 2.1 considering $s_1 \leq_c s_2 \leq_c s_3$, the next event of signal s_2 after the $2^n d$ informative event of s_3 will be:

$$nextEvent(s_2, e_1(s_3)) = e_3(s_2)$$

but the next event of s_3 after the second informative event of s_2 will be:

$$nextEvent(s_3, e_1(s_2)) = e_1(s_3)$$

Definition 2.5.12. *Given a behavior $b = \{s_1, \ldots, s_j, \ldots, s_N\}$ and an event $e_k(s_j) = (v_k, t_k)$, a stall move returns a behavior $b' = stall(e_k(s_j)) = \{s_1, \ldots, s'_j, \ldots, s_N\}$, s.t for all $l \in N$*

$$\begin{aligned} \sigma_{[t_0,t_{k-1}]}(s'_j) &= \sigma_{[t_0,t_{k-1}]}(s_j) \\ \sigma_{[t_k,t_k]}(s'_j) &= \tau \\ \sigma_{[t_{k+l+1},t_{k+l+1}]}(s'_j) &= \sigma_{[t_{k+l},t_{k+l}]}(s_j). \end{aligned}$$

Example 2.10. In figure 2.2(a), b is a behavior with three signals s_1, s_2 and s_3. Now if we insert a stall at $e_1(s_1)$ then the corresponding $b' = stall(e_1(s_1))$ is shown if figure 2.2(b).

$$b = \begin{cases} s_1 = & i_1, i_2, \tau, i_3, i_4, i_5, i_6, i_7, \tau, \tau, \tau, \tau, \ldots \\ s_2 = & i_1, i_2, \tau, i_3, i_4, i_5, \tau, i_6, i_7, \tau, \tau, \tau, \ldots \\ s_3 = & i_1, i_2, \tau, i_3, \tau, i_4, i_5, \tau, i_6, \tau, i_7, \tau, \ldots \end{cases}$$

(a)

$$b' = \begin{cases} s_1 = & i_1, \tau, i_2, \tau, i_3, i_4, i_5, i_6, i_7, \tau, \tau, \tau, \ldots \\ s_2 = & i_1, i_2, \tau, i_3, i_4, i_5, \tau, i_6, i_7, \tau, \tau, \tau, \ldots \\ s_3 = & i_1, i_2, \tau, i_3, \tau, i_4, i_5, \tau, i_6, \tau, i_7, \tau, \ldots \end{cases}$$

(b)

Figure 2.2: *An example of $stall$ function*

2.5.2 Procrastination effect

A *procrastination effect* represent the "effect" of a stall move $stall(e_k(s_j))$ on other signals of behavior b in correspondence of events following $e_k(s_j)$ in the lexicographic order. The process will "respond" to the insertion of stalls in some of their signals by "delaying" other signals that are causally related to the stalled signals. Given a behavior b for each stall move on event of b, we have a corresponding set of behaviors (the procrastination effect set)

Definition 2.5.13. *Procrastination in is a point-to-set map that takes a behavior $b' = (s'_1, , s'_N) = stall(e_k(s_j))$ resulting from the application of a stall move on event $e_k(s_j)$ of behavior $b = (s_1, , s_N)$ and returns a set of behaviors $\mathcal{PE}[stall(e_k(s_j))]$ s.t. $b'' = (s''_1, , s''_N) \in \mathcal{PE}[b']$ iff the following three conditions hold*

1. $s''_j = s'_j$;

2. $\forall i \in [1, N], i \neq j, s_i'' \equiv_\tau s_i'$ *and* $\sigma_{[t_0,t_{l-1}]}(s_i'') = \sigma_{[t_0,t_{l-1}]}(s_i')$ *where* t_l *is the time stamp of event* $e_l(s_i) = nextEvent(s_i, e_k(s_j))$.

3. $\exists K$ *finite* $s.t. \forall i \in [1, N], i \neq j, \exists k_i \leq K, \sigma_{[t_{l+k_i},\infty]}(s_i'') = \sigma_{[t_l,\infty]}(s_i')$

Each behavior in $\mathcal{PE}[b']$ *is obtained from* b' *by possibly inserting other stalling events in any signal of* b'*, but only at "later timestamps. Procrastination effect returns a behavior that latency dominates the original behavior.*

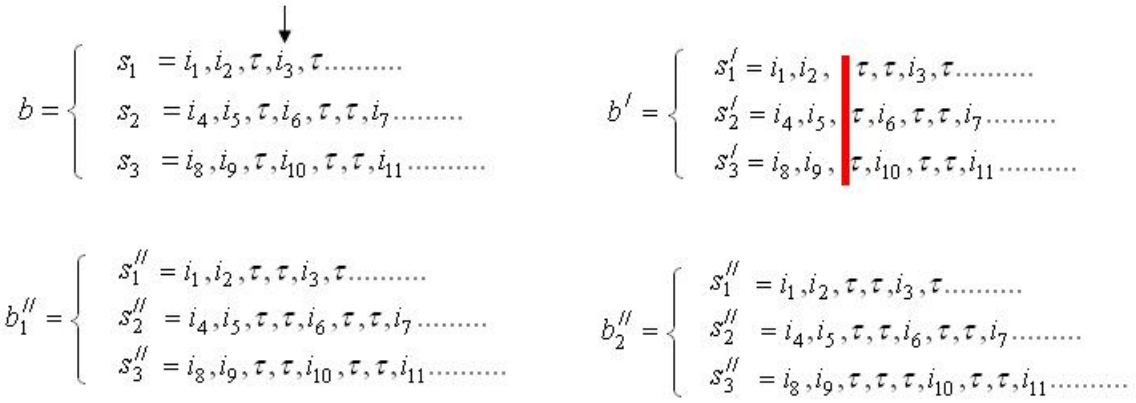

Figure 2.3: *An example of procrastination effect.*

Example 2.11. Let us consider the b given in figure 3.1 as a behavior of a process P. Now if we want to insert a delay at $e_3(s_1)$ as shown by an arrow in the figure then according to definition of $stall(e_3(s_1))$, a new behavior b' will produce. This new behavior is shown in the figure as b'. We can see that the information i_3 which was in the 3^{rd} event (starting from zero) of s_1 in b is now shifted to the 4^{th} event of signal s_1' in b'. The remaining two signals of b' are same as their corresponding signals in b. Consider that $s_1 \leq_c s_2 \leq_c s_3$. In b, i_1 is happening before i_4 and i_4 is happening before i_8. The same condition is true for i_2, i_5, i_9 and i_3, i_6, i_{10} also. If we observe b' carefully we can see that the causality relation among the events of all the signals is satisfying up to the red line. But after the red line causality relation is not satisfying (i_3 is not happening before i_6 and i_{10}). Due to this reason procrastination effect is necessary to produce a behavior b'' from b' in such a way that it should satisfy the causality relation of P. We can do it by inserting some delays in s_2' and s_3' as shown

in the figure by b_1'' and b_2''. Both b_1'' and b_2'' are satisfying the above three condition of procrastination effect.

2.5.3 Patient processes and their Compositionality

The key condition in the LI design method is that the intellectual property (IP) blocks must be patient. A patient process can take stall moves on any signal of its behaviors by reacting with the appropriate procrastination effects.

Definition 2.5.14. *A process P is a patient process iff*

$$\begin{aligned} \forall b = (s_1, \ldots, s_N) \in P, \forall j \in [1, N] \\ \forall e_k(s_j) \in \mathcal{E}_\iota(s_j) \\ ((\mathcal{PE}[stall(e_k(s_j))] \cap P \neq \emptyset). \end{aligned}$$

Hence a process is called patient process if it is possible to insert any number of delays in any of its signal without changing any property of the process. The explanation of the above mathematical definition of patient process is as follows: If we apply a delay in a signal which belongs to a behavior b of the process P then according to the definition 2.5.12 and the definition of procrastination effect a set of N-tuples will produce which will satisfy the three conditions of procrastination effect. This set is represented by $\mathcal{PE}[stall(...)]$. All the elements in this set will maintain the causality relation of P but it does not mean that all the elements will also satisfy the properties of P. And we know that if any N-tuple satisfies the properties of a process P then the tuple is called a behavior of P. If any N-tuple from the give set of N-tuples, is a behavior of P then it can be say that after inserting the delay it is possible to maintain the original properties of the process. If these is possible for inserting delays in any signal belongs to any behavior of P, then P is said to be a patient process.

Each module in the LI design must be patient. And the procedure to convert each IP core as a patient process is already discussed in chapter 1. But we have to clear some confusion yet. If all the modules in a SoC are patient then what is the guarantee that the whole system will be patient? This is the most important question because LI

design always consider that the system must be patient. Carloni et el. [2] proposed some theorems to prove that the composition of patient processes is also a patient process. The theorems are explained below:

Lemma 2.5.1. *Let P_1 and P_2 be two patent processes. Let $b_1 \in P_1, b_2 \in P_2$ be two behaviors with the same lexicographic order s.t. $b_1 \equiv_\tau b_2$. Then, there exist a behavior $b' \in (P_1 \cap P_2), b_1 \equiv_\tau b' \equiv_\tau b_2$.*

Explanation 2.5.1. The proof of this lemma is given in [2]. Given that b_1 is latency equivalent with b_2 but this does not mean that b_1 and b_2 are always equal. When b_1 and b_2 are equal, b' will just same as b_1 and b_2 because $b' \in (P_1 \cap P_2)$. So definitely $b_1 \equiv_\tau b' \equiv_\tau b_2$. But when b_1 and b_2 are just latency equivalent, it is not possible to say directly that b' is latency equivalent with both b_1 and b_2. Fortunately there is a procedure to produce a behavior $b_1^* \in P_1$ from b_1 and a behavior $b_2^* \in P_2$ from b_2, s.t. $b_1^* \equiv_\tau b_1$ and $b_1^* \equiv_\tau b_1$ and $b_1^* = b_2^*$. In this case we can say that $b' = b_1^* = b_2^*$ and hence we can also say that $b_1 \equiv_\tau b' \equiv_\tau b_2$.
The figure 2.4 shows an example for explaining the procedure by which b_1^* and b_2^* are producing. Let b_1 and b_2 shown in the figure 2.4(a) be the behavior of P_1 and P_2 respectively. Now to produce b_1^* and b_2^* from b_1 and b_2, s.t. $b_1^* = b_2^*$, we have to proceed step by step. In each step we align the first unaligned pair of the corresponding events.

In step 1 the first unaligned pair of corresponding events is $pir_1 = [e_1(s_1 \in b_1), e_3(s_2 \in b_2)]$. These two events are shown by blue arrows in the figure 2.4(a). To align these two events we have to shift the event $[e_1(s_1 \in b_1)]$ by two position. The only way to do this is to insert delays at $[e_1(s_1 \in b_1)]$. After inserting one delay at $[e_1(s_1 \in b_1)]$ we got a behavior $b_1'' \in (\mathcal{PE}(stall(e_1(s_1 \in b_1))) \cap P_1)$. Figure 2.4(b) showing the newly created b_1'' with the original b_2. Now comparing b_1'' and b_2 we can see that the two events of pir_1 are still not aligned. So we insert another delay at $[e_2(s_1'' \in b_1'')]$. As a result we got a behavior $b''' \in (\mathcal{PE}(stall(e_2(s_1'' \in b_1''))) \cap P_1)$ as shown in figure 2.4(c). Now we can see that the two events of pir_1 are aligned with each other. So step 1 finish.

In step 2, the first unaligned pair from figure 2.4(c) is $pir_2 = [e_4(s_2 \in b'''_1), e_3(s_2 \in b_2)]$. For aligning them we have to insert one delay at $[e_4(s_2 \in b'''_1)]$. The resulting behavior $b_2'' \in (\mathcal{PE}(stall(e_3(s_2 \in b_2'')))\cap P_2)$ is shown in the figure 2.4(d). Comparing the two behavior b_1''' and b_2'' we can see that the two unaligned events of pair pir_2

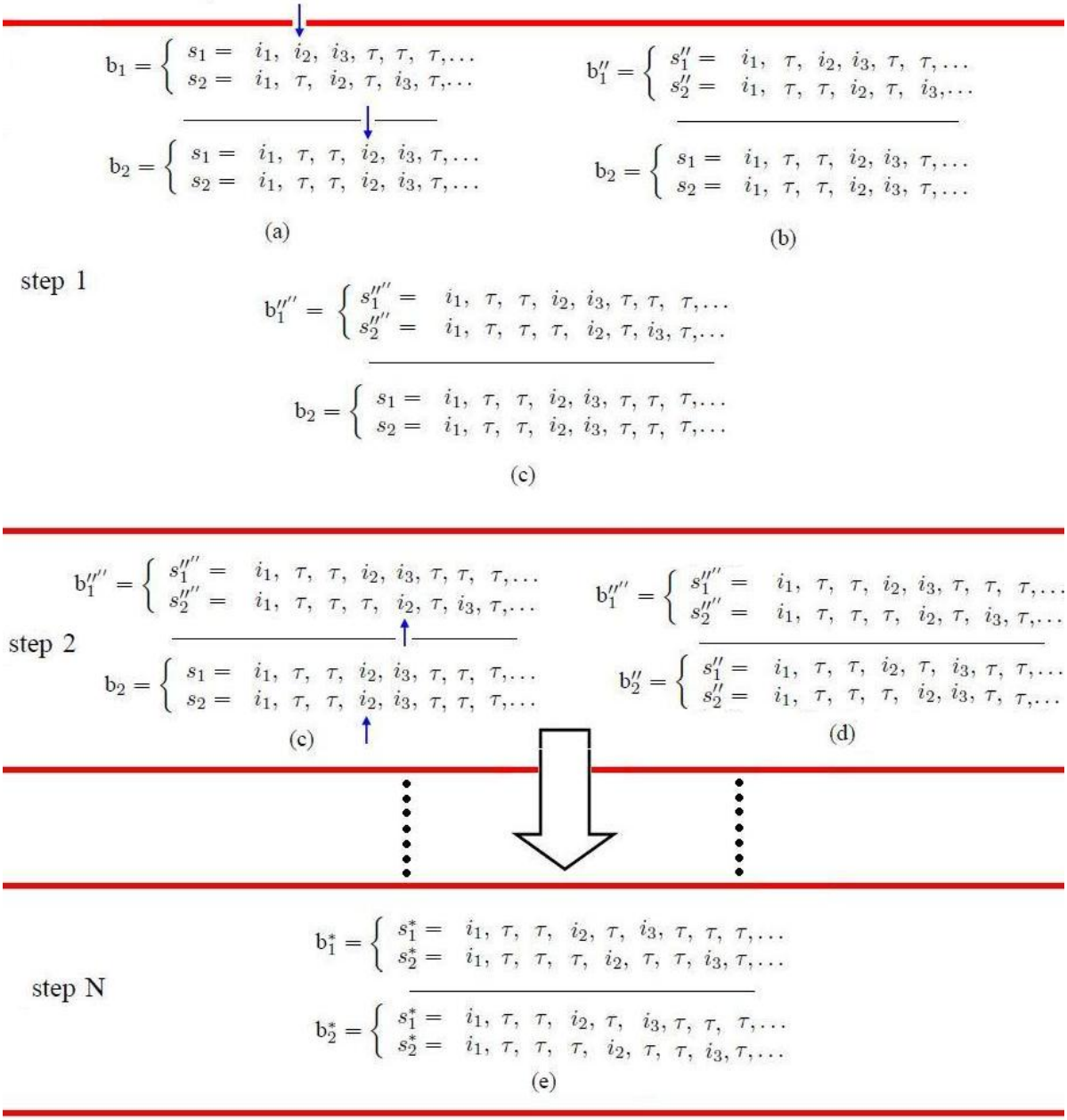

Figure 2.4: *An example to explain Lemma 2.5.1*

has been aligned. So step 2 finish.

In this way within a finite number of steps, it is possible to align all the corresponding events of b_1 and b_2. But since we are aligning by inserting delays, hence according to procrastination effect and the properties of patient process finally we get two behaviors b_1^* and b_2^* s.t. $b_1^* = b_2^*$ and $b_1^* \equiv_\tau b_1$, $b_2^* \equiv_\tau b_2$ and $b_1^* \in P_1$, $b_2^* \in P_2$.

Theorem 2.5.2. *If P_1 and P_2 are two patient processes, then $(P_1 \cap P_2)$ is also a patient process.*

Explanation 2.5.2. The proof of this theorem is given in [], here we are giving a explanation of the proof. Let $P = (P_1 \cap P_2)$, now to proof that P is a patient process, we have show the following:

- *"for any behavior $b \in P$, if we insert a delay in any of its signal (say z) then we have to got a behavior $\hat{b}$ s.t. $\hat{b} \in \mathcal{PE}(stall(z))$ and $\hat{b} \in P$."*

Let $b \in P, b_1 \in P_1$ and $b_2 \in P_2$ are the behaviors of P, P_1 and P_2 respectively. Consider also that $b_1 = b_2$. Now according to the lemma 2.5.1, $b \equiv_\tau b_1 \equiv_\tau b_2$. Let the signal where we want to inset the delay is s_j and the event is $e_k(s_j)$. Note that if $s_j \in P$ then it must be true that $s_j \in P_1$ and $s_j \in P_2$.

Now let us consider that after inserting a delay at $e_k(s_j)$ we get a behavior b'' s.t. $b'' \in \mathcal{PE}(stall(e_k(P[s_j])))$. Similarly let $b_1'' \in \mathcal{PE}(stall(e_k(P_1[s_j])))$ and $b_2'' \in \mathcal{PE}(stall(e_k(P_2[s_j])))$. Also consider that $b_1'' \in P_1$ and $b_2'' \in P_2$. It is not always true that $b'' = b_1'' = b_2''$. But applying the procedure given in the explanation of lemma 2.5.1, it is possible to find three behaviors b^*, b_1^* and b_2^* from b'', b_1'' and b_2'' respectively, s.t. $b^* = b_1^* = b_2^*$. Now since $b_1^* \in P_1$ and $b_2^* \in P_2$ it can be proof that $b^* \in P$. But to prove P as a patient process we still need to show that $b^* \in \mathcal{PE}(stall(e_k(P[s_j])))$. If we observe the procedure of getting b^* from b'' we can see that b^* is obtained by just inserting delays after the $nextEvent(P(s_i''), e_k(P[s_j'']))$, where s_i'' stands for all the signals in b''. So it means that $b^* \in \mathcal{PE}(stall(e_k(P[s_j])))$. Hence $P = (P_1 \cap P_2)$ is a patient process.

Theorem 2.5.3. *For all process P_1, P_2, P_1', P_2', if $P_1 \equiv_\tau P_1'$ and $P_2 \equiv_\tau P_2'$ then $P_1 \cap P_2) \equiv_\tau (P_1' \cap P_2')$.*

Theorem 2.5.4. *For all strict processes P_1, P_2 and all patient processes P_1', P_2', if $P_1 \equiv_\tau P_1'$ and $P_2 \equiv_\tau P_2'$, then $(P_1 \cap P_2) \equiv_\tau (P_1' \cap P_2')$.*

The proof of the above two theorem are given in [2].

From the above theorems we can say that it is possible to replace all the processes in a system of strict processes by corresponding patient process and the resulting system will be patient process and latency equivalent to the original one. This is the core idea of LI design: take a design based on the assumption that computation in one functional module and communication among modules "take no time" (synchronous hypothesis), i.e., the process corresponding to the functional modules and their composition are strict and replace it with a design where communication does take time and as a result, signals are delayed, but without changing the sequence of informative events observed at the system level.

2.5.4 Patient Process and Relay Stations

We already mentioned that LI design is based on combining the IP cores. Each IP core must make patient and if all the modules (patient IP cores) in the system are patient then the entire system will also patient. But actually in LI design relay station are inserted into the channels to divide them into segments. So we cannot just say that a system designed by LI design contains only patient IP cores but it also contains relay stations. On other words a system designed by LI design is a combination of patient IP cores and relay stations. Now it clear that to be the system patient the relay stations must also be patient. In this section we will describe relay station's and there properties.

2.5.4.1 Channel and Buffers

The tagged signal model provides the notion of channel to formalize the composition of processes [**?**]. A channel is a connection constraining two signals to be identical.

Definition 2.5.15. *A channel* $\mathcal{C}(i,j) \subset S^N, i,j \in [1,N]$ *is a process, s.t.*

$$b = (s_1, \ldots, s_N) \in \mathcal{C}(i,j) \Leftrightarrow s_i = s_j.$$

A channel is not a patient process [2] because it lacks the capacity of storing an event and delaying its communication between two processes. Hence to formally

model communication delays as well as pipeline stages, Carloni et el. [2] introduce the notion of buffer. A buffer is a process relating two signals s_i and s_j of a behavior b and is defined by means of three parameters: capacity c, minimum forward latency l_f, and minimum backward latency l_b. A buffer forces signals s_i, S_j to be latency equivalent and to satisfy the following relationships for all natural numbers k.

- The difference between the amount of informative events seen at s_i from timestamps zero to timestamps $k - l_f$ and the amount of informative events seen at s_j from timestamps zero to timestamps k is greater or equal than zero.
- The difference between the amount of informative events seen at s_i from timestamps zero to timestamps k and the amount of informative events seen at s_j from timestamps zero to timestamps $k - l_b$ is at most c.

Definition 2.5.16. *A buffer $\mathcal{B}^c_{l_f,l_b}(i,j)$ with capacity $c \geq 0$, minimum forward latency $l_f \geq 0$, and minimum backward latency $l_b \geq 0$ is a process s.t. $\forall i,j \in [1,N] : b = (s_1,\ldots,s_N) \in \mathcal{B}^c_{l_f,l_b}(i,j)$ iff $(s_i \equiv_\tau s_j)$ and $\forall k \in N$*

$$0 \leq |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k-l_f)}]}(s_i)]| - |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)]| \tag{2.3}$$

$$c \geq |\mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)]| - |\mathcal{F}_\iota[\sigma_{[t_0,t_{(k-l_b)}]}(s_j)]| \tag{2.4}$$

By definition, given a pair of indexes $i,j \in [1,N]$, for all $l_b, l_f, c \geq 0$, all buffers $\mathcal{B}^c_{l_f,l_b}(i,j)$ are latency equivalent. Note that $\mathcal{B}^0_{0,0}(i,j)$ is nothing but a channel. LI design requires patient buffers having unitary latencies. The following theorem proves that a buffer with unitary latencies is a patient process.

Theorem 2.5.5. *Let $l_b = l_f = 1$. For all $c \geq 1, \mathcal{B}^c_{l_f,l_b}(i,j)$ is patient iff $s_i \leq_c s_j$.*

Explanation 2.5.3. The proof of this theorem is given in [2], we are here explaining the proof with some examples.
Only if: if $s_i \not\leq_c s_j$, then $\mathcal{B}^c_{1,1}(i,j)$ will never be a patient process. For example consider a bhavior b of the buffer $\mathcal{B}^2_{1,1}(i,j)$ as

$$b = \begin{cases} s_i = & i_0, i_1, \tau, \tau, \ldots \\ s_j = & \tau, i_0, i_1, \tau, \ldots \end{cases}.$$

Now if we insert a delay at $e_0(s_i)$, then the according to the function $stall(e_0(s_i))$ we will get a b' as follows:

$$b' = \begin{cases} s'_i = & \tau, i_0, i_1, \tau, \tau, \dots \\ s'_j = & \tau, i_0, i_1, \tau, \dots \end{cases}.$$

Definitely b' will not satisfy the equation 2.3 and 2.4. Now it is possible to show that all $b'' \in \mathcal{PE}(stall(e_0(s_i))$, will also not satisfy the equation 2.3 and 2.4. Since $s_i \not\leq_c s_j$ hence all the $b'' \in \mathcal{PE}(stall(e_0(s_i))$ will be as follows:

$$b'' = \begin{cases} s''_i = & \tau, i_0, \dots, \dots \\ s''_j = & \tau, i_0, \dots, \dots \end{cases}.$$

Definitely b'' is not satisfying equation 2.3 and 2.4. Hence $b'' \not\in \mathcal{B}^2_{1,1}(i,j)$

If: if $s_i \leq_c s_j$ then $\mathcal{B}^2_{1,1}(i,j)$ is patient process. According to the definition of patient process we have to show that:

1. After inserting a delay at any possition (let k) of signal s_i there must be a $b'' \in \mathcal{PE}[stall(e_k(s_i))]$, which satisfy the properties of buffer, i.e. equation 2.3 and 2.4.

2. After inserting a delay at any possition (let k) of signal s_j there must be a $b'' \in \mathcal{PE}[stall(e_k(s_j))]$, which satisfy the properties of buffer, i.e. equation 2.3 and 2.4.

below we are explaining the above two conditions with example

1. To explane the first condition we need to consider two examples. The examples are shown below as case1 and case 2 respectively.
   Case1: Let us consider a behavior $b \in \mathcal{B}^2_{1,1}(i,j)$ as shown below. We can verify that the behavior b satisfies the equation 2.3 and 2.4.

$$b = \begin{cases} s_i = & i_0, i_1, \tau, \tau, \dots \\ s_j = & \tau, \tau, i_0, i_1, \tau, \dots \end{cases}.$$

   Now if we apply $stall(e_1(s_i))$ then the corresponding b' will be as follows:

$$b' = \begin{cases} s'_i = & i_0, \tau, i_1, \tau, \tau, \dots \\ s'_j = & \tau, \tau, i_0, i_1, \tau, \dots \end{cases}.$$

Clearly b' satisfying equation 2.3 and 2.4. Hence $b' \in \mathcal{B}^2_{1,1}(i,j)$.

Case2: Let us consider a behavior $b \in \mathcal{B}^2_{1,1}(i,j)$ as shown below. We can verify that the behavior b satisfies the equation 2.3 and 2.4.

$$b = \begin{cases} s_i = & i_0, i_1, \tau, \tau, \ldots \\ s_j = & \tau, \tau, i_0, i_1, \tau, \ldots \end{cases}.$$

Now if we apply $stall(e_3(s_j))$ then the corresponding b' will be as follows:

$$b' = \begin{cases} s'_i = & i_0, i_1, \tau, \tau, \ldots \\ s'_j = & \tau, \tau, i_0, \tau, i_1, \tau, \ldots \end{cases}.$$

Clearly b' is not satisfying equation 2.3. Hence $b' \notin \mathcal{B}^2_{1,1}(i,j)$. For proving $\mathcal{B}^2_{1,1}(i,j)$ is patient we have to find one behavior $b'' \in \mathcal{PE}(b')$, which satisfies the equation 2.3 and 2.4. Only then we can say that $b'' \in \mathcal{B}^2_{1,1}(i,j)$. The following behavior shows one such b'' which belongs to $\mathcal{B}^2_{1,1}(i,j)$ and also satisfying the equation 2.3 and 2.4.

$$b'' = \begin{cases} s''_i = & i_0, \tau, i_1, \tau, \tau, \tau, \ldots \\ s''_j = & \tau, i_0, \tau, i_1, \tau, \tau, \ldots \end{cases}.$$

Hence in case of condition 1, $\mathcal{B}^2_{1,1}(i,j)$ is a patient process.

2. To explane the second condition we also need to consider two examples. The examples are shown below as case1 and case 2 respectively.

   Case1: Let us consider a behavior $b \in \mathcal{B}^2_{1,1}(i,j)$ as shown below. We can verify that the behavior b satisfies the equation 2.3 and 2.4.

$$b = \begin{cases} s_i = & i_0, i_1, i_2, \tau, \tau, \ldots \\ s_j = & \tau, i_0, i_1, i_2, \tau, \ldots \end{cases}.$$

   Now if we apply $stall(e_1(s_j))$ then the corresponding b' will be as follows:

$$b' = \begin{cases} s'_i = & i_0, i_1, i_2, \tau, \tau, \ldots \\ s'_j = & \tau, \tau, i_0, i_1, i_2, \tau, \ldots \end{cases}.$$

   Clearly b' satisfying equation 2.3 and 2.4. Hence $b' \in \mathcal{B}^2_{1,1}(i,j)$.

   Case2: Let us consider a behavior $b \in \mathcal{B}^2_{1,1}(i,j)$ as shown below. We can verify that the behavior b satisfies the equation 2.3 and 2.4.

$$b = \begin{cases} s_i = & i_0, i_1, i_2, \tau, \tau, \ldots \\ s_j = & \tau, i_0, i_1, i_2, \tau, \ldots \end{cases}.$$

Now if we apply $stall(e_1(s_j))$ then the corresponding b' will be as follows:

$$b' = \begin{cases} s'_i = & i_0, i_1, i_2, \tau, \tau, \dots \\ s'_j = & \tau, \tau, i_0, i_1, i_2, \tau, \dots \end{cases} .$$

Clearly b' is not satisfying equation 2.4. Hence $b' \notin \mathcal{B}^2_{1,1}(i,j)$. For proving $\mathcal{B}^2_{1,1}(i,j)$ is patient we have to find one behavior $b'' \in \mathcal{PE}(b')$, which satisfies the equation 2.3 and 2.4. Only then we can say that $b'' \in \mathcal{B}^2_{1,1}(i,j)$. The following behavior shows one such b'' which belongs to $\mathcal{B}^2_{1,1}(i,j)$ and also satisfying the equation 2.3 and 2.4.

$$b'' = \begin{cases} s''_i = & i_0, i_1, \tau, i_2, \tau, \tau, \dots \\ s''_j = & \tau, \tau, i_0, i_1, i_2, \tau, \dots \end{cases} .$$

Hence in case of condition 2, $\mathcal{B}^2_{1,1}$ is a patient process.

Definition 2.5.17. *The buffer $\mathcal{B}^2_{1,1}$ is called a relay station.*

Now it can be say that the combination of all patient modules and the relay stations in a LI designed system will also be a patient system. The details about how the signals are handled by the modules as well as the relay station is explained in [2].

Chapter 3

Proposed Theory

3.1 From Single Clock to Multiple Clock LI design

In this section, we discuss the differences between the single clock and multiple clock systems. We start by listing the elements in the proof of correctness of LI design and see how each of them change because of the paradigm shift. Later in this section, we discuss the need for new definitions (if any) and their implications on the multiple clock LI design.

The elements in the theory of LI design are as follows.

- *Tagged Signal Model* - The traditional LI design is applicable only for single clock systems, where all the modules have the same clock frequency. In the tagged signal model, this corresponds to the fact that the set of tags for all the signals are same. However, in the multiple clock systems, the whole system is divided into synchronous blocks and these synchronous blocks communicate with each other using asynchronous or synchronous protocols. In the tagged signal model, this implies that the set of signals are classified into several equivalence classes and signals in the same equivalence class are synchronous with each other. Thus, in order to define an interconnect between two signals that do not belong to the same equivalence class, we need new semantics, that should be able to adequately represent the system.

- *Patient process and Procrastination Effect* - Procrastination effect represents the "effect" of a stall move $stall(e_k(s_j))$ on other signals in the behavior b

in correspondence to the event $e_k(s_j)$ in lexicographical order. The difference between the single clock and multiple clock designs is that, in multiple clock systems, one time period of a clock need not be equal to the time period of another clock. Thus a stall signal $stall(e_k(s_j))$ will have a different type of effect on the module than that is described in the procrastination effect in the single clock systems. We may therefore need to modify the definition of procrastination effect for multiclock processes.

- *Relay station* - In the single clock LI design, the relay station is an interconnect which connects two modules where the delay induced because of the wire may lead to synchronization errors. However in the multiple clock design there is one more issue: there is a difference in the clock frequencies of the two connecting modules. In this case even if there is no considerable delay induced by the wire, the difference of clock frequencies will lead to synchronization errors. Thus, in order to create a multiple clock latency insensitive relay station, we need to take care of two things:

  1. The synchronization error caused due to latency
  2. The synchronization error caused due to difference of clock frequencies or phase differences using same frequency.

  With changes in the tagged signal model, we give a denotational model for a relay station that handles the second type of synchronization errors. Later in section 3.5, we present an approach where both these synchronization errors can be handled.

- *Stallable functional modules* - The stallable functional modules are synchronous functional modules around which a wrapper is built in order to make them latency insensitive. In the case of multiple clock systems, these functional modules remain to be synchronous and thus there is no change in them. All modifications are done to the interconnect.

From the above discussion, we infer the following:

1. The wrapper that makes functional module latency insensitive in the traditional LI design is latency insensitive in the new domain too.

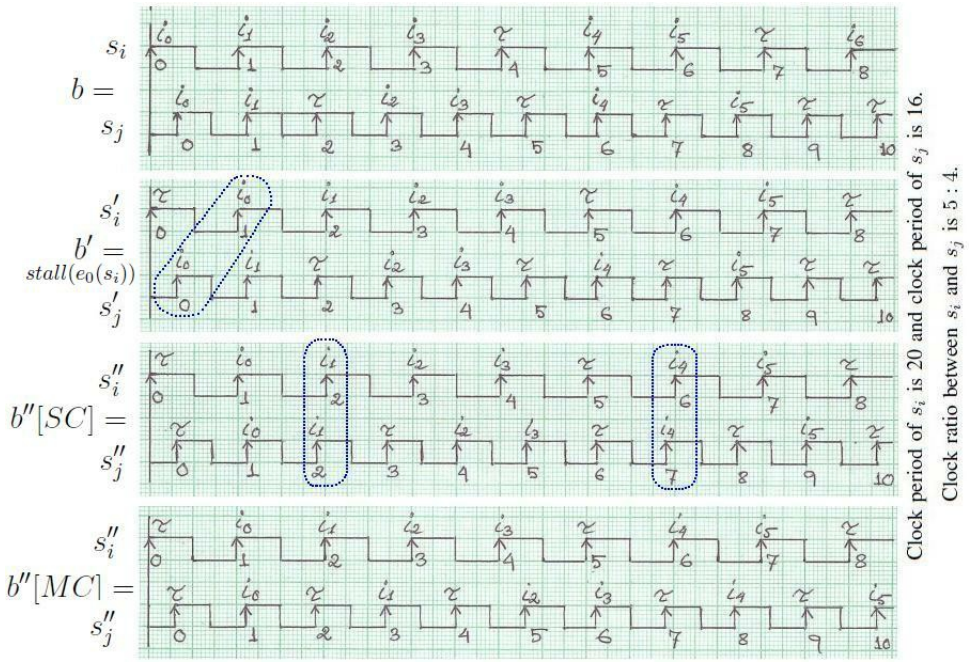

Figure 3.1: *Example showing the necessity to change the procrastination effect definition.*

2. The changes in the tagged signal model and the need for multiple clock interconnect require new semantics and new definitions.

3. The jump from single clock to multiple clock will need a new definition of procrastination effect for the multiple clock processes.

Below in example 3.2, we show one illustration, where the difference in clock frequency requires changes in procrastination effect and the need for new semantics for multiple clock interconnect.

Definition 3.1.1. *Clock ratio* $n : m$ *between the two signals* s_i *and* s_j *means the ratio of their clock periods. If* p_i *is the clock period of* s_i *and* p_j *is the clock period of* s_j *then* $n : m = p_i : p_j$.

□

3.2 A Basic Technique for MCLI Design

In this section we discuss a basic technique of MCLI design which we proposed in our first stage of project. The method is based on LCM and has some limitations. In this design technique, we proposed to design an MCLI system with different single-clock modules. The technique was as follows:

1. Collect some single clock IP cores (not necessarily belongs to same clock).
2. Encapsulate them with a wrapper circuit to make them patient as explained in 1.1.2.
3. Design the system without considering the delays.
4. Insert single clock relay station [2], into the channels between two synchronous modules where communication required more than one clock cycle.
5. Insert multiple clock relay station (MRS) (A relay station proposed by us), where communication is required between two domains not synchronous to each other. This relay station inputs data from one frequency domain and forwards it to another frequency domain; as well as it can tolerate back-pressure.

The main property of LI systems is that it can handle any arbitrary delays in any of its channel, without violating any of its property. Each single-clock component (IP modules and Single Clock Relay Stations) can handle delays according to the techniques proposed in [2]. We just need to handle delays in our proposed multiple clock relay stations. Hence, to handle delay issues created by input delays or back-pressure, the MRS has to insert stalls in various connecting signals. The number of stalls to be inserted depends on the ratio of the clock frequencies to be connected. The main observation being that the events in two sequences have co-relation every $LCM(p_i, p_j)$ time units, where p_i and p_j are the clock periods of two connecting channels. The following example gives a detail explanation about how MRS handles delays inserted into its channels.

Example 3.1. In figure 3.2 we have taken a behavior b containing two signals s_i and s_j with a clock ratio 5:4 and $s_i \leq_c s_j$. The grid structure in the background of the figure indicates the real time value of an event $(\mathcal{V}_E)$ as defined in the next subsection.

We numbered each events of the signals as $0, 1, 2...N$. The value shown at the top of every tick indicates the information at that event. We also numbered all the informative events of a signal sequentially. This numbering has no relation with event numbering, e.g. i_1 does not means that it is the information at 1^{st} event. Note that this condition is true for the figures **??**, **??**, **??** and **??** also. Note the correspondence between informative events in s_i and s_j. The informative events in s_j follow after the corresponding one in s_i. Now if we apply $stall(e_0(s_i))$, we get behavior b'. We can see from the figure that the $\leq_{lo}$ relation of b is not properly maintaining by b'. The problematic area is shown by dotted eclipse. According to [2] in a single clock system, this situation can be handled by inserting a delay at $e_0(s_j)$. The resultant b'' is shown in the figure as $b''[SC]$. As can be seen, the order is maintained for i_0 but is no longer valid for i_1 and i_4 of s_i (shown by dotted eclipse). Since the two signals s_i and s_j are not synchronous, the $b''[SC]$ still not completely satisfying the causality relation of b. But it is possible to maintain the same correlation between the events by shifting both the event $e_0(s_i)$ and $e_0(s_j)$ by the LCMP amount of time, where LCMP means the LCM of the two clock periods. For this we need to insert 4 stalls in $e_0(s_i)$ and 5 stalls in $e_0(s_j)$. So we handle the delays in MRS, by delaying both the signals with the LCMP amount of time. The final solution is shown in figure as $b''[MC]$.

We have provided denotational semantics for this technique and also proved that the technique is correct in terms of LI design properties. The details can be found in our **MTP first-stage Project Report**. But later we found that the technique is not efficient and it is doing unnecessary delays in the system. Though, it is correct but it has some limitations.

3.2.1 Limitation With This Technique

The main limitation is the delay with LCM time. Due to this the MRS may delay unnecessarily for some time. This unnecessary delays will degrade the performance of the entire system because this delays will propagate to other modules in the system. The primary goal of the MRS is to transfer data from one clock domain to another domain. Hence instead of delaying LCM amount of time to maintain the correlation between the signals, the MRS can just concentrate on sending data after receiving it.

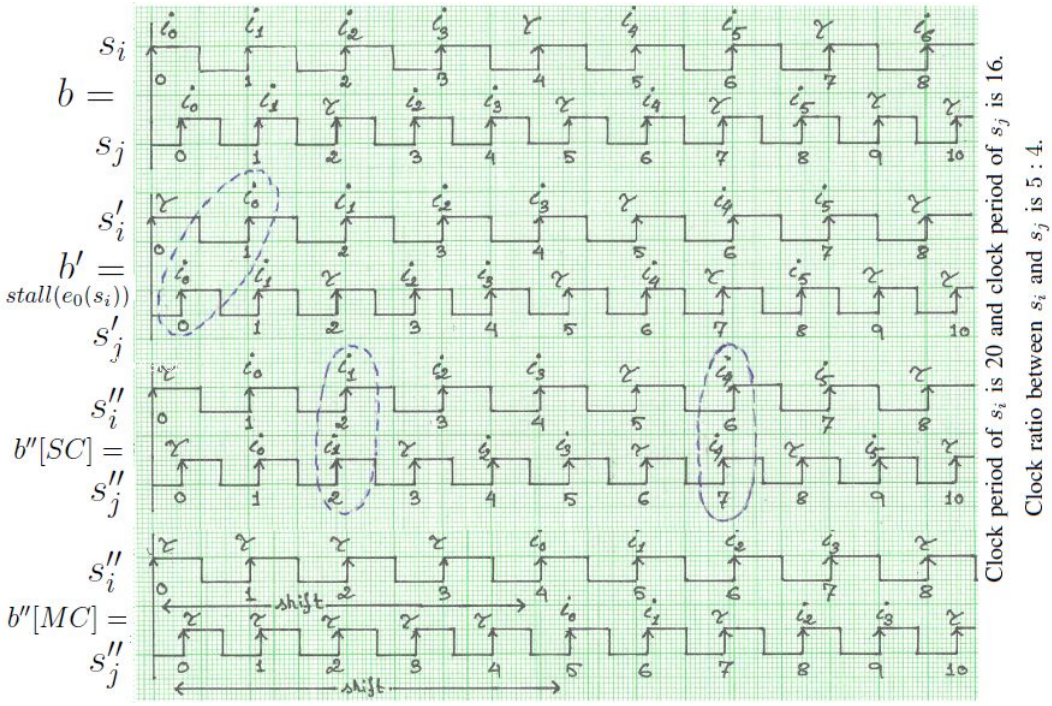

Figure 3.2: *Example showing the necessity to change the procrastination effect definition.*

In this way the MRS needs to delay with the minimum amount of time. Hence, the performance of the entire MCLI system will increase.

Due to this reason, we propose a new technique for designing the MCLI systems. Henceforth, all the discussions, in this report are based on our new design technique.

3.3 An Example of Handling Delays in Our New Technique

Example 3.2. In figure 3.1 we have taken a behavior b containing two signals s_i and s_j with a clock ratio 5:4 and $s_i \leq_c s_j$. The grid structure in the background of the figure indicates the real time value of an event $(\mathcal{V}_E)$ as defined later in section 3.4. We numbered each events of the signals as $0, 1, 2...N$. The value shown at the top of every tick indicates the information at that event. We also numbered all the informative events of a signal sequentially. This numbering has no relation with event

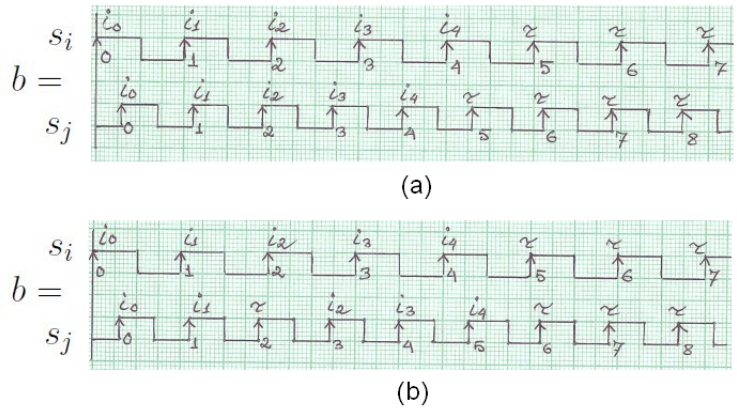

Figure 3.3: *Example showing two behaviors (a) Strict behavior and (b) Not a strict behavior.*

numbering, e.g. i_1 does not mean that it is the information at first event. Also note the correspondence between informative events in s_i and s_j. The informative events in s_j follow (in real time) after the corresponding one in s_i.

Now if we apply $stall(e_0(s_i))$, we get behavior b'. We can see from the figure that the causality relation of b is not properly maintained by b'. The problematic area is shown by dotted eclipse. According to [2] in a single clock system, this situation can be handled by inserting a delay at $e_0(s'_j)$. The resultant b'' is shown in the figure as $b''[SC]$. As can be seen, the order is maintained for i_0 but is no longer valid for i_1 and i_4 (shown by dotted eclipse). Since the two signals s_i and s_j are not synchronous, $b''[SC]$ is still not completely satisfying the causality relation of b.

So to solve this problem we need to check signals s_i and s_j completely and insert stalls in conflicting places. The final solution is shown in figure as $b''[MC]$ which we will discuss later in this chapter. □

3.3.1 Some important concepts about multiple clock LI design

As we mentioned earlier that we have used most of the definitions and terms used in [2]. But since we are talking about multiple clock systems, some confusion arise about the correctness of the old definitions in the new domain. In this section we discuss how earlier definitions are still valid.

Strict processes in multiple clock domain- As mentioned in [2], a signal is called

a strict signal if and only if all informative events precede all stalling events. And a process is called a strict process if it contains only strict signals. For example the following is a strict process having two signals s_i and s_j.

$$P = \begin{cases} s_i = & i_1, i_2, i_3, i_4, i_5, i_6, i_7, \tau, \tau, \tau, \ldots \\ s_j = & i_1, i_2, i_3, i_4, i_5, i_6, , \tau, \tau, \tau, \ldots \end{cases}$$

The main concept of single clock LI design is to initially assume the system without any communication delays. In this case all processes in the system are strict processes. If any communication delay occurs during the implementation, then it must not violate the functionality of the system. For this reason, all strict processes are replaced by patient processes. And the composition of all patient processes will remain patient.

However in the case of a multiple clock LI design we cannot assume a "zero communication delay" system as a strict system. For example consider a process P (see figure 3.3(a)) having two signals s_i(input) and s_j(output) with different clock periods. Signal s_j outputs information after receiving it from s_i. Now from the figure 3.3(a) we can see that it is not possible for a strict behavior of P to satisfy the input-output relationship of the process.

Therefore in the case of multiple clock LI design a strict system is assumed to run on an imaginary (virtual) clock that handles the synchronization delays to gives a strict behavior. This essentially defines the functionality of the system without delay/synchronization errors. After implementing with actual clocks, all the functionalities of the strict process (with the imaginary clock), must be maintain.

For this reason we need to replace all strict processes with multiple clock patient processes. We will show that the composition of all these multiple clock patient processes is also a multiple clock patient process.

Equivalence of two signals- In multiple clock systems two signals s_i and s_j having different clock periods are said to be equal if $\forall l \in \mathbb{N}$ and $\forall k \in \mathbb{N}$ if $ord(e_l(s_i)) = ord(e_k(s_j))$ then $l = k$. It means that we are not considering the actual clock periods during the signal comparison. We are doing it by comparing the clock ticks. In figure 3.3(a) the two signals s_i and s_j are equal.

Equivalence of two behaviors- In multiple clock system two behaviors are said to be equivalent or aligned if their corresponding signals are equal.

Lexicographic Order- There is no deference between the definition of lexicographic order in multiple clock and in the single clock domain. The reason is that we are comparing events with their clock tick numbers and not with the real clock times. The ordinal of the behavior b given in the figure 3.3(a) (assuming $s_i \leq_c s_j$) will be as follows: $e_0(s_i), e_0(s_j), e_1(s_i), e_1(s_j), e_2(s_i), e_2(s_j), e_3(s_i), e_3(s_j),$ $e_4(s_i), e_4(s_j)$

From the above ordering of b given in the figure 3.3(a) we can see $e_2(s_i) <_{lo} e_2(s_j)$, but due to smaller clock period of s_j, $e_2(s_j)$ occurred earlier than $e_2(s_i)$. So a question may arise about the correctness of the definition of lexicographic order in multiple clock environment. The problem is handling by the properties of the process. Each process has some properties and a N-touple (set of N signals) can only be a behavior of a process if it satisfies the properties of the process. So if we consider a process P with two signals s_i and s_j having property that s_i and s_j must maintain the input-output relationship (i.e.,s_j can only send an informative event after receiving it from s_i), then the N-touple b given in the figure 3.3(a) is not a behavior of process P. Figure 3.3(b) showing a correct behavior of process P.

3.4 New definitions required for multiple clock LI design

In the previous section we have explained and illustrated the need for new semantics and new definitions so that multiple clock LI design can be modeled in tagged signal framework. In this section, we give the new definitions.

Definition 3.4.1. *The mapping $\mathcal{V}_T : \mathcal{T} \rightarrow \mathbb{R}$ from the set of tags to the real number domain is called the time-map of the events if the following properties hold for signal s_i and s_j*

1. $\forall s_i$, *if* $e_0(s_i) = (v_{i_0}, t_{i_0})$ *then* $\mathcal{V}_T(t_{i_0}) = 0$
2. *If* $e_1(s_i) = (v_1, t_1)$ *and* $e_k(s_i) = (v_k, t_k)$, *then* $\mathcal{V}_T(t_k) = k \times \mathcal{V}_T(t_1)$
3. *If s_i and s_j are not synchronous signals, and* $e_0(s_i) = (v_{i_0}, t_{i_0})$ *and* $e_0(s_j) = (v_{j_0}, t_{j_0})$ *then* $\mathcal{V}_T(t_{i_0})$ *may not be equal to* $\mathcal{V}_T(t_{j_0})$.

□

The mapping $\mathcal{V}_E : \{\Sigma \cup \{\tau\}\} \times \mathcal{T} \rightarrow \mathbb{R}$ is defined such that for an event $e_k = (v_k, t_k)$, we have $\mathcal{V}_E(e_k) = \mathcal{V}_T(t_k)$.

Definition 3.4.2. *For every signal s_i, since $\mathcal{V}_E(e_k(s_i)) = k \times \mathcal{V}_E(e_1(s_i))$, we define the clock period of a signal s_i as $\mathcal{V}_E(e_1(s_i))$.* □

Since there is a difference in the clock frequencies, in order to maintain a timeline of the system, the mapping for every event in a signal across the system to the set of natural numbers $\mathbb{N}$ is not enough. If we need to express a multiple clock interconnect, we need stricter semantics which can relate events in two signals that are not synchronous. This is the reason why we need a mapping from each event of a signal to the real numbers $\mathbb{R}$, which will denote the timeline.

We use the mapping from the signals to $\mathbb{R}$, to relate the events in two signals that are not synchronous with each other. We now define the *prevTimingEvent*. This definition of *prevTimingEvent* will be useful in defining the interconnect between two different clock frequencies.

Definition 3.4.3. *Consider a channel $C(i, j)$, where the two signals s_i and s_j fall into two different equivalence classes of signals. Now we say $e_l(s_j) = prevTimingEvent(s_j, e_k(s_i))$ iff*

$$l = (k)_i^j = max\{t | \forall t \in N, \mathcal{V}_E(e_t(s_j)) < \mathcal{V}_E(e_k(s_i))\} \tag{3.1}$$

□

We denote the *prevTimingEvent* of k^{th} event of signal i in signal j as $(k)_i^j$.

Example 3.3. In figure 3.3(b), the *prevTimingEvent* of $e_2(s_i)$ in s_j is $e_2(s_j)$, i.e. $prevTimingEvent(s_j, e_2(s_i)) = e_2(s_j)$. Similarly, $prevTimingEvent(s_i, e_2(s_j)) = e_1(s_i)$. □

If we carefully observe the definitions of $nextEvent$ [2] and *prevTimingEvent*, the former is only concerned about the $order$ of the events and the $\leq_c$ relation. It will therefore appear in all the definitions which are concerned with the latency equivalence of signals and procrastination effect. Also, it is worth noticing that the $nextEvent$

is only applicable for informative events. However the *prevTimingEvent* event is only concerned about the timing and the relation between the timing sequences of events. Since it is purely a relation between timing of various events, it can be applied to any event both informative as well as stall events.

As we have mentioned already that the lexicographic order ($\leq_{lo}$) of a behavior b, is an ordered set of informative events in b. This ordering depends on the ordinal and the clock ticks of the signals. It has no relation with the real time values of the clock. But in case of multiple clock processes, we also require an ordering, which can order the events of its behaviors in terms of real time clock values. Here we define an ordering which orders the events of a behavior, according to the real time clock values.

Definition 3.4.4. *Given a behavior $b = (s_1, \ldots, s_N) \in P$, where P is a multiple clock process. The $realTimeOrdering(b)$ gives an ordered set X for all the events, $x_i (i \in \mathbb{N})$, in b such that x_{i+1} follows x_i iff*

$$\forall i \in \mathbb{N}, \forall x_i \in X : (V_E(x_i) < V_E(x_{i+1})) \vee$$

$$(V_E(x_i) = V_E(x_{i+1}) \wedge (s_k \leq_c s_l)).$$

Where s_k and s_l are the signals containing the events x_i and x_{i+1} respectively. □

Example 3.4. In figure 3.3(a), the *realTimeOrdering* of the behavior b is as follows:
realTimeOrdering(b)= $\{e_0(s_i), e_0(s_j), e_1(s_i), e_1(s_j),$
$e_2(s_j), e_2(s_i), e_3(s_j), e_3(s_i), e_4(s_j), e_4(s_i), e_5(s_j), \ldots\}$ □

Now we are giving a general definition for *process*, considering both single clock and multiple clock domains.

Definition 3.4.5. *A process is called m-process if it has signals of m different clock frequencies, where $m > 0$. So a synchronous process can also be termed as a 1-process. Note that process working on same clock but having phase difference is treated as a multiple clock process, i.e. it has $m = 2$.* □

We now define the procrastination effect for the *m-process*. The procrastination effect represents the "effect" of a stall move $stall(e_k(s_j))$. As illustrated in the

previous section, we have noticed that introducing stall events in a signal j will "effect" other signals by introducing several lexicographic stall events following $e_k(s_j)$ (cf. $b''[SC]$ in Fig. 3.1). But since there is a difference of clock frequencies ($m > 1$), a stall event in a signal corresponds to zero/one/more stalls in other signals. It is because of this reason, we need to introduce zero or more stalls even in the original signal s_j at the events that will come after $e_k(s_j)$ lexicographically. We formalize this definition as given below:

Definition 3.4.6. *Procrastination effect for m-process is a point-to-set map that takes a behavior* $b' = (s'_1, , s'_N) = stall(e_k(s_j))$ *resulting from the application of a stall move on event* $e_k(s_j)$ *of behavior* $b = (s_1, , s_N)$ *and returns a set of behaviors* $\mathcal{MPE}[stall(e_k(s_j))]$ *st* $b'' = (s''_1, , s''_N) \in \mathcal{MPE}[b']$ *iff the following two conditions hold*

1. $\forall i \in [1, N], s''_i \equiv_\tau s'_i$ *and* $\sigma_{[t_0,t_{l-1}]}(s''_i) = \sigma_{[t_0,t_{l-1}]}(s'_i)$ *where* t_l *is the time stamp of event* $e_l(s_i) = nextEvent(s_i, e_k(s_j))$.

2. *if* $m = 1$ *then,*

   a) $s''_j = s'_j$.

   b) $\exists K$ *finite s.t.* $\forall i \in [1, N], i \neq j, \exists k_i \leq K, \sigma_{[t_{l+k_i},\infty]}(s''_i) = \sigma_{[t_l,\infty]}(s'_i)$.

   else if $m > 1$ *then,*

   c) $\exists K$ *finite s.t.* $\forall i \in [1, N]$ *and* $\forall x \in [1, \infty], \exists k_{ix}$ *s.t.* $0 \leq k_{ix} \leq K, \sigma_{[t_{l^x+sum_i+k_{ix}},t_{l^{x+1}-1+sum_i+k_{ix}}]}(s''_i) = \sigma_{[t_{l^x},t_{l^{x+1}-1}]}(s'_i)$.
   Where $sum_i = \left\{ \begin{array}{ll} \sum_{y=1}^{x-1} k_{iy} & if\ x > 1 \\ 0 & if\ x = 1 \end{array} \right\}$
   and l^x *means the* x*th* $nextEvent(s_i, e_k(s_j))$, *e.g.* l^1 *means* $nextEvent(s_i, e_k(s_j))$ *and* l^2 *means* $nextEvent(s_i, l^1)$. *In other words,* l^x *means* x*th informative event of* s_i *after the event* $e_k(s_j)$ *in lexicographic order.*

□

Observe that if $m = 1$ then the $\mathcal{MPE}(b')$ will exactly behave like the procrastination effect defined in [2]. But if $m > 1$ then the condition 2(c) needs some explanation. In simple words the meaning of this condition is, if $m > 1$ then the set of behaviors produced by $\mathcal{MPE}(b')$ is obtained from b' by inserting zero or more stalls

in place of every informative events, which succeed $e_k(s_j)$ in lexicographic order. Here i indicates signal and x indicates the informative events of signal s_i which succeed $e_k(s_j)$ lexicographically. Hence k_{ix} means a variable for the xth informative event from the set of all informative events of s_i, which succeeds $e_k(s_j)$ lexicographically. Now to obtain b'' from b', we need to insert k_{ix} delays in place of all informative events which succeed $e_k(s_j)$ lexicographically, where $0 \leq k_{ix} \leq K$. The expression

$$\sigma_{[t_{l^x}+sum_i+k_{ix}, t_{l^{x+1}-1}+sum_i+k_{ix}]}(s_i'') = \sigma_{[t_{l^x}, t_{l^{x+1}-1}]}(s_i')$$

seems to be very complex, but its meaning is very simple. Here we explain the expression. As we already explain the meaning of l^x, we can say that, there must be only one informative event in each of the sequences $\sigma[t_{l^1}, t_{l^2-1}](s_i')$,$\sigma[t_{l^2}, t_{l^3-1}](s_i')$ and so on. b'' cannot have any extra informative events which is not in b', because $b' \equiv_\tau b''$. Here b'' can only have extra stalling events than b'. As we mentioned that b'' produces from b' by inserting zero or more stalls at the place of every informative events (of any signal) which succeed $e_k(s_j)$ lexicographically. Now if we insert k_{i1} stall at $e_{l^1}(s_i')$ to obtain b'' then each event in s_i' starting from $e_{l^1}(s_i')$ will be shifted by k_{i1}. Hence all the events in the sequence $\sigma[t_{l^1}, t_{l^2-1}](s_i')$ will be shifted by k_{i1} and the corresponding sequence will be $\sigma[t_{l^1+k_{i1}}, t_{l^2-1+k_{i1}}](s_i'')$. Clearly, $\sigma[t_{l^1+k_{i1}}, t_{l^2-1+k_{i1}}](s_i'') = \sigma[t_{l^1}, t_{l^2-1}](s_i')$. Here $sum_i = 0$ because $x = 1$. Now if we insert k_{i2} stalls at $e_{l^2}(s_i')$ then each event in s_i' starting from $e_{l^2}(s_i')$ will be shifted by $k_{i1} + k_{i2}$. Hence clearly, $\sigma[t_{l^2+k_{i1}+k_{i2}}, t_{l^3-1+k_{i1}+k_{i2}}](s_i'') = \sigma[t_{l^2}, t_{l^3-1}](s_i')$. Here $sum_i = k_{i1}$, because $x = 2$. In this way we can show the equality for all the corresponding sequences between b' and b'', for all i and all x.

Note that the condition $s_j'' = s_j'$, which is true for single clock procrastination effect ($\mathcal{PE}$) in [2] is not necessary here when $m > 1$. This reflects the fact that because of implicit difference in clock frequencies, a stall event in a signal will have a complex effect over the behavior of the entire system and this correctly represents the multiple clock domain.

Example 3.5. In figure 3.1 the process we have taken is a *2-process*. And the $b''[MC]$ we get from b' is by applying $\mathcal{MPE}(b')$ as follows:

- inserting 0 stall in s_i'.
- inserting 1 stall at $e_0(s_j')$.

- inserting 1 stall at $e_1(s'_j)$.
- inserting 0 stall in place of all other informative events of s'_j.

□

3.4.1 Compositionality of multiple clock patient process

We now come to the core of multiple clock LI design, i.e. patient process. This definition is dependent on the multiple clock procrastination effect.

Definition 3.4.7. *An m-process P is a multiple clock patient process iff*

$$\forall b = (s_1, \ldots, s_N) \in P, \forall j \in [1, N]$$
$$\forall e_k(s_j) \in \mathcal{E}_\iota(s_j)$$
$$((\mathcal{MPE}[stall(e_k(s_j))] \cap P \neq \emptyset)$$

□

If a process is a patient process according to [2], then the process is also a multiple clock patient process. The reason behind this is all patient processes according to [2] are actually *1-process*, so $m = 1$ and according to our definition of multiple clock procrastination effect, the set of b'' produced when $m = 1$ is same as the set of b'' produced by the procrastination effect of [2].

Below we prove some Lemma's and Theorem's regarding the composition of multiple clock patient processes. The proves are almost same as that given in [2]. We write them here for completeness purpose. The only difference being in the use of multiple clock procrastination effect ($\mathcal{MPE}$) instead of procrastination effect ($\mathcal{PE}$) from [2].

Lemma 3.4.1. *Let P_1 and P_2 be two multiple clock patient processes. Let $b_1 \in P_1, b_2 \in P_2$ be two behaviors with the same lexicographic order s.t. $b_1 \equiv_\tau b_2$. Then, there exists a behavior $b' \in (P_1 \cap P_2), b_1 \equiv_\tau b' \equiv_\tau b_2$.*

Proof: The proof for this Lemma is almost same as that given in [2]. The $\mathcal{MPE}$ will introduce more stalls than those by $\mathcal{PE}$ and hence the substeps of each alignment step will increase the stalls but only by a finite amount.

Consider that $b_1 = (r_1, \ldots, r_N) \in P_1$ and $b_2 = (q_1, \ldots, q_N) \in P_2$ are the two behaviors having same lexicographic order. Since b_1 and b_2 are latency equivalent, each event in b_1 has a corresponding event in b_2 and vice versa. Let $r_1 \equiv_\tau q_1, \ldots, r_N \equiv_\tau q_N$. Let $W = \{w | \forall j \in [1, N], \exists k \in N, \exists l \in N (k \neq l \ \wedge ord(e_k(r_j)) = ord(e_l(q_j)) = w)\}$ be the set of ordinals associated to pairs of corresponding events of b_1 and b_2 whose timestamps differ. We define the distance between the two behavior to be zero if $W = \emptyset$.

$$d(b_1, b_2) = 0, \quad \text{if } W = \emptyset.$$

Thus, if all pairs of corresponding events in b_1 and b_2 are aligned then $d(b_1, b_2) = 0$. It means that the behaviors b_1 and b_2 are identical and they are the same behavior which belongs to $(P_1 \cap P_2)$. Now in case of $d(b_1, b_2) \neq 0$, w_1 is the smallest ordinal among those which are associated to unaligned pairs of corresponding events. If more than one pairs have the smallest ordinal then take the pair having the event which lexicographically precedes all the remaining events of the pairs. Let $p_1 = (e_k(r_j), e_l(q_j))$ be the pair of corresponding events having ordinal w_1 chosen considering the above mentioned rule and let $k < l$. Now insert a stall at $e_k(r_j)$ to get a new behavior $b_1' = (s_1', \ldots, s_N') = stall(e_k(r_j)) \equiv_\tau b_1$. Obviously, $slack(e_{k+1}(s_j'), e_l(q_j)) = slack(e_k(r_j), e_l(q_j)) - 1$. Note that b_1' may not be a behavior of P_1. But, since P_1 is a multiple clock patient process, there exists a behavior $b_1'' = (s_1'', \ldots, s_N'') \equiv_\tau b_1$ s.t. $b_1'' \in \mathcal{MPE}(stall(e_k(r_j))) \cap P_1$. Since, multiple clock procrastination effect is defined for both single clock and multiple clock processes we do not need to take care about which type of process it is. According to the definition of the multiple clock procrastination effect $slack(e_{k+1}(s_j''), e_l(q_j)) = slack(e_k(r_j), e_l(q_j)) - 1$. Since the procrastination effect cannot postpone the events preceding $e_k(r_j)$ in the lexicographic order $\leq_{lo}$, hence all the pairs of corresponding events of b_1'' and b_2'' with ordinal smaller than w_1 are still aligned. Now, there are two possibilities: if $slack(e_{k+1}(s_j''), e_l(q_j)) = 0$, then it means that one more pair has been aligned and $d(b_1'', b_2) < d(b_1, b_2)$. The other possibility is if $slack(e_{k+1}(s_j''), e_l(q_j)) \neq 0$, in this case we have to repeat the same procedure starting from the behavior b_1''. In this way,

after $l-k$ steps of the above mentioned procedure, we will get a behavior $b_1^* \equiv_\tau b_1$ that satisfies P_1 and s.t. $d(b_1^*, b_2) < d(b_1, b_2)$ because one more pair of corresponding events has been aligned. Let us say that we have performed an *alignment step*.

Now, if the distance between b_1^* and b_2 become zero i.e. $d(b_1^*, b_2) = 0$, then there are no more unaligned pairs and the behaviors are identical. In this case the lemma is proved because $b_1^* \equiv_\tau b_1$. But if the distance between b_1^* and b_2 is not equal to zero i.e. $d(b_1^*, b_2) \neq 0$, then we have to align the next unaligned pair p_2 of corresponding events. For this we need to execute the alignment step again. It is possible that at the mth step, while aligning the pair p_m with ordinal w_m, we may increase the slack of some of the pairs following p_m in the lexicographic order, but we keep aligned all the pairs preceding p_m. During this sequence of alignment steps, we obtain two sequences of behaviors (one from P_1 latency equivalent to b_1 and one from P_2 latency equivalent to b_2) whose distance is decreasing monotonically.

Let U be a the set of pairs of unaligned corresponding events. Now we know that $b_1 \equiv_\tau b_2$, it means that both b_1 and b_2 contains same number of informative events and it is finite. In this case, the set U also contains finite number of unaligned pairs. The slack of each of these pair is also a finite number. Consider that at the mth step, we have at most h_m substeps to align p_m, where h_m is the starting slack for p_m. According to the definition of our multiple clock procrastination effect (see Definition 3.4.6), getting a behavior (b'') from $\mathcal{MPE}(b')$ need to insert finite number of stalls in b'. Let this finite number as X. Hence in the worst case, each behavior b^* obtained during the substeps of the alignment step may have slacks of all the remaining unaligned pairs increased by at most X. Now after the end of mth step, $|U|$ has been decreased by one, while all the slacks of its remaining elements have been increased by at most $h_m.X$, which is a finite number. Hence, for $|U| \geq l \geq m$, the new slacks for the remaining unaligned pairs is $h'_l \leq h_l + h_m.X$. Globally, in the worst case we have to perform $|U|$ alignment steps and for each of them we have a finite number of substeps. Therefore, the two sequences of behaviors are also finite and the last elements of both sequences do not have unaligned pairs. It means that the last elements of both sequences are identical and having zero distance. □

Theorem 3.4.2. *If P_1 and P_2 are multiple clock patient processes, then $(P_1 \cap P_2)$ is also a multiple clock patient process.*

Proof: Let $b = (r_1, \ldots, r_N)$ be a behavior in $P_1 \cap P_2$. Also consider two behaviors $b_1 = (x_1, \ldots, x_N) \in P_1$ and $b_2 = (y_1, \ldots, y_N) \in P_2$, s.t. $b_1 = b_2 = b$. Now for all $j \in [1, N]$ and for all $k \in N$, let $e_k(r_j) \in \mathcal{E}_l(r_j)$. Now since $b_1 = b_2 = b$, then $e_k(x_j) \in \mathcal{E}_\iota(x_j)$ and $e_k(y_j) \in \mathcal{E}_\iota(y_j)$. Consider $b' = stall(e_k(r_j)) \equiv_\tau b$ and $b'_1 = stall(e_k(x_j)) \equiv_\tau b_1$. Similarly, $b'_2 = stall(e_k(y_j)) \equiv_\tau b_2$. Since P_1 is a multiple clock patient process, there exists a behavior $b''_1 \equiv_\tau b_1$ s.t. $b''_1 \in \mathcal{MPE}(b'_1) \cap P_1$ and since P_2 is a multiple clock patient process, there exists a behavior $b''_2 \equiv_\tau b_2$ s.t. $b''_2 \in \mathcal{MPE}(b'_2) \cap P_2$. Now $b_1 = b_2$ implies that $b''_1 \equiv_\tau b''_2$, but it is not necessary that $b''_1 = b''_2$. The reason is that the multiple clock procrastination effect may have misaligned the pairs of corresponding informative events that come after $(e_k(x_j), e_k(y_j))$ with respect to lexicographic order $\leq_{lo}$. Since $b_1 = b_2$ share the same lexicographic order, by Lemma 3.4.1, there exists a behavior $b'' \equiv_\tau b''_1 \equiv_\tau b''_2$ s.t. $b'' \in P_1 \cap P_2$. From the proof of Lemma 3.4.1 we can see that the construction of b'' involves only unaligned pairs of corresponding events between b''_1 and b''_2. All these unaligned pairs correspond to informative events that come after $e_k(r_j)$ with respect to the lexicographic order $\leq_{lo}$. Now, since the number of informative events is finite and the number of unaligned pairs is also finite. Hence, each signal r''_i of b'' is obtained by inserting a finite number of stalling events not earlier than $e_l(r_i) = nextEvent(r_i, e_k(r_j))$. Therefore, by Definition 3.4.6, $b'' \in \mathcal{MPE}(stall(e_k(r_j)))$. Since we have already seen that $b'' \in P_1 \cap P_2$, then $(P_1 \cap P_2)$ is a multiple clock patient process. □

Theorem 3.4.3. *For all multiple clock patient processes P_1, P_2, P'_1, P'_2, if $P_1 \equiv_\tau P'_1$ and $P_2 \equiv_\tau P'_2$ then $(P_1 \cap P_2) \equiv_\tau (P'_1 \cap P'_2)$.*

Proof: Consider that $b = (r_1, \ldots, r_N)$ be a behavior in $P_1 \cap P_2$. According to the definition of latency equivalence we can say that there must be behaviors $b_1 = (x_1, \ldots, x_N) \in P'_1$ and $b_2 = (y_1, \ldots, y_N) \in P_2$ such that $b_1 \equiv_\tau b \equiv_\tau b_2$. Now since b_1 and b_2 are latency equivalent and P'_1 and P'_2 are multiple clock patient processes, according to Lemma 3.4.1 we can say that there must be a latency equivalent behavior $b' \in (P'_1 \cap P'_2)$. The other direction of the proof is symmetric. □

Theorem 3.4.4. *For all strict process P_1, P_2 and all multiple clock patient processes P'_1, P'_2, if $P_1 \equiv_\tau P'_1$ and $P_2 \equiv_\tau P'_2$, then $(P_1 \cap P_2) \equiv_\tau (P'_1 \cap P'_2)$.*

Proof: According to Theorem 3.4.3 every behavior in $(P_1 \cap P_2)$ has an equivalent behavior in $(P_1' \cap P_2')$. Now let b' be a behavior in $(P_1' \cap P_2')$. According to the definition of latency equivalence we can say that that there must be behaviors $b_1 \in P_1$ and $b_2 \in P_2$ such that $b_1 \equiv_\tau b' \equiv_\tau b_2$. Since P_1 and P_2 are strict, b_1 and b_2 are also strict. Now due to latency equivalent, they must, therefore be equal. Hence, $b_1 \in (P_1 \cap P_2)$. □

This means that it is possible to replace all processes in a system of strict processes by corresponding multiple clock patient processes and the resulting system will be latency equivalent to the original one.

3.5 Multiple clock patient process and relay station

The main subject of this work is to model and verify the multiple clock LI design. The steps for the same are given in Section 3.1. In this section we give the formal definition of the multiple clock buffer and multiple clock relay station with the assumption under which the relay station is multiple clock patient.

3.5.1 Channels and Single clock Buffers

The notion of channel is provided in the tagged signal model to formalize the composition of processes [**?**]. A channel is a connection constraining that two signals to be identical. Hence channel always have a single clock.

Definition 3.5.1. *A channel* $C(i,j) \subset S^N, i,j \in [1,N]$ *is a process s.t.* $b = (s_1, \ldots, s_N) \in C(i,j) \Leftrightarrow s_i = s_j$. □

From [2] we know a channel is not a single clock patient process. Now the following lemma formally proves that the channel is also not a multiple clock patient process.

Lemma 3.5.1. *A channel* $C(i,j) \subset S^N$ *is not a multiple clock patient process.*

Proof: It has already proved in [2] that channel is not a single clock patient process. Now according to the definition of multiple clock procrastination effect, when $m = 1$ i.e. when the process has only one clock, then for any stall (say at event

x of s_i), $\mathcal{MPE}(stall(x))$ will always give exactly the same set of behaviors as those given by $\mathcal{PE}(stall(x))$, where $\mathcal{PE}$ is the single clock procrastination effect defined in [2]. Therefore if a process has only one clock and it is not a single clock patient process then it cannot be a multiple clock patient process either. Since channel $C(i,j)$ having only one clock and it is not a single clock patient process hence it cannot be a multiple clock patient process. □

By single clock buffer we exactly mean the buffer as defined in [2]. It is a process relating two signals s_i and s_j of a behavior b and is defined by means of three parameters: capacity c, minimum forward latency l_f, and minimum backward latency l_b. The definition of single clock buffer as in [2] is given.

Definition 3.5.2. *A buffer $\mathcal{B}^c_{l_f,l_b}(i,j)$ having capacity $c \geq 0$, minimum forward latency $l_f \geq 0$, and minimum backward latency $l_b \geq 0$ is a process s.t. $\forall i,j \in [1,N] : b = (s_1,\ldots,s_N) \in \mathcal{B}^c_{l_f,l_b}(i,j)$ iff $(s_i \equiv_\tau s_j)$ and $\forall k \in N$*

$$0 \leq \mathcal{F}_\iota[\sigma_{[t_0,t_{k-l_f}]}(s_i)] - \mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)] \tag{3.2}$$

$$c \geq \mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)] - \mathcal{F}_\iota[\sigma_{[t_0,t_{k-l_b}]}(s_j)] \tag{3.3}$$

□

Now we need to prove that a single clock buffer $\mathcal{B}^c_{l_f,l_b}(i,j)$ having $c \geq 1, l_f = 1$ and $l_b = 1$ can be a multiple clock patient process.

Theorem 3.5.2. *Let $l_b = l_f = 1$. For all $c \geq 1, \mathcal{B}^c_{1,1}(i,j)$ is multiple clock patient iff $s_i \leq_c s_j$.*

Proof: From [2] we know that the above mentioned singl clock buffer $\mathcal{B}^c_{1,1}(i,j)$ is a single clock patient process. Now from the proof of Lemma 3.5.1 we know that if a process having only one clock is not a single clock patient process, then it cannot be a multiple clock patient process and if a process having only one clock is a single clock patient process then it must also be a multiple clock patient process. Hence clearly we can say that $\mathcal{B}^c_{1,1}(i,j)$ is a multiple clock patient process. □

3.5.2 Single clock Relay Stations

Single clock relay station means the relay station defined in [2]. The definition of single clock relay station is reproduced here.

Definition 3.5.3. *The single clock buffer $\mathcal{B}^2_{1,1}(i,j)$ is called a single clock relay station.* □

The reason of taking the capacity of the single clock relay stations as 2 is already explained in [2].

3.5.3 Multiple clock Buffers

As pointed out in section 3.1 we need a new definition for multiple clock relay station. In this section, we define a buffer which describes the behavior of two signals s_i and s_j which are not synchronous with each other. Our primary goal is to define an interconnect which takes care of the synchronization errors generated by the difference in clock frequencies including phase difference using same frequency. We then give a method by which we can generate relay station which will handle the latency induced by wire as well as the synchronization error due to difference in clock frequencies. The multiple clock buffer is a process relating two signals s_i and s_j of a behavior b. Its difference with single clock buffer is that the two signals s_i and s_j have different clock frequencies. So a multiple clock buffer receives informations at the speed of s_i and sends them at the speed of s_j, provided s_i be the input signal and s_j is the output signal. It is defined by means of two parameters: capacity c and the *prevTimingEvent* [see definition 3.4.3]. A multiple clock buffer forces signals s_i, s_j to be latency equivalent and to satisfy the following relationships for all natural numbers k.

1. The difference between the amount of informative events seen at s_j from event number zero to event number k and the amount of informative events seen at s_i from event number zero to the *prevTimingEvent* of $e_k(s_j)$ in s_i is less than equal to zero. In other words s_j does not send (new) events which s_i has not received.

2. The difference between the amount of informative events seen at s_i from the event number zero to event number k and the amount of informative events seen at s_j between event number zero to the *prevTimingEvent* of $e_k(s_i)$ in s_j is at most c. In other words, event not yet sent are stored in the buffer.

Definition 3.5.4. *A multiple clock buffer* $\mathcal{B}^c_{mck}(i,j)$ *with capacity* $c \geq 1$*, where* s_i *and* s_j *are not synchronous with each other and* mck *denotes the multiple clock domain, is a process s.t.* $\forall i,j \in [1,N] : b = (s_1,\ldots,s_N) \in \mathcal{B}^c_{mck}(i,j)$ *iff* $s_i \equiv_\tau s_j$ *and* $\forall k \in \mathbb{N}$

$$0 \geq \mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_j)] - \mathcal{F}_\iota[\sigma_{[t_0,t_{(k)^i_j}]}(s_i)] \tag{3.4}$$

$$c \geq \mathcal{F}_\iota[\sigma_{[t_0,t_k]}(s_i)] - \mathcal{F}_\iota[\sigma_{[t_0,t_{(k)^j_i}]}(s_j)] \tag{3.5}$$

□

Theorem 3.5.3. *For all* $c \geq 1, \mathcal{B}^c_{mck}(i,j)$ *is a multiple clock patient process.*

Proof: We need to prove that if we insert a stall at any signal (s_i or s_j) of a behavior b of $\mathcal{B}^c_{mck}(i,j)$, then according to the multiple clock procrastination effect we must get a behavior b^* which satisfies the inequality 3.4 and 3.5.

Only if: By contradiction, we prove that if $s_i \not\leq_c s_j$, then $\mathcal{B}^c_{mck}(i,j)$ is not a multiple clock patient process. Suppose that $s_i \not\leq_c s_j$, let $b = (s_1,,s_N)$, be a behavior of $\mathcal{B}^c_{mck}(i,j)$ s.t ratio of time periods of $s_i : s_j = 2 : 1$ and $\sigma(s_i) = \iota_0, \iota_1,$ and $\sigma(s_j) = \tau, \iota_0, \tau, \iota_1,$. Let $b' = (s'_1,,s'_N) = stall(e_0(s_i))$ with $e_0(s_i) = (\iota_0, t_0)$. Clearly $b' \notin \mathcal{B}^c_{mck}(i,j)$ because inequality 3.4 doesn't hold for $k = 0$. Further for all $b'' = (s_1,,s_N) \in \mathcal{MPE}[stall(e_0(s_i))]$, $b'' \in \mathcal{B}^c_{mck}(i,j)$ iff $\sigma_{[t_1,t_1]}(s''_j) = \tau$. However , consider that $ord(e_0(s_i)) = ord(e_1(s_j))$ and since $s_i \not\leq_c s_j$ then $e_0(s_i) \not\leq_{lo} e_1(s_j)$. Further $ord(e_0(s_i)) = ord(e_3(s_j)) - 1$.
Thus $e_3(s_j) = nextEvent(s_j, e_0(s_i))$. Recall that by definition of procrastination effect , $\sigma_{[t_0,t_{l-1}]}(s''_j) = \sigma_{[t_0,t_{l-1}]}(s'_j)$ where t_l is the time stamp of $nextEvent(s_j, e_0(s_i))$ and in this case, $t_l = t_3$ and thus $\sigma_{[t_0,t_2]}(s''_j) = \sigma_{[t_0,t_2]}(s'_j)$. Since $s'_j = s_j$, we get that $\sigma_{[t_1,t_1]}(s''_j) = \sigma_{[t_1,t_1]}(s'_j) = \sigma_{[t_1,t_1]}(s_j) = \iota_0 \neq \tau$. Thus $b'' \notin \mathcal{B}^c_{mck}(i,j)$. Hence $\mathcal{MPE}[stall(e_0(s_i))] \cap \mathcal{B}^c_{mck}(i,j) = \emptyset$ and $\mathcal{B}^c_{mck}(i,j)$ is not patient.

If: We prove that if $s_i \leq_c s_j$ then $\mathcal{B}^c_{mck}(i,j)$ is a patient process. This proof involves analyzing three cases where stalls are introduced at three different channels: on s_i, s_j and s_r with $r \in ([1,N]/\{i,j\})$. For the remaining part of the proof let T_i and T_j be the clock periods of s_i and s_j respectively.

Case 1: For all $g \in \mathbb{N}$, such that $e_g(s_i) \in \mathcal{E}_\iota(s_i)$, let $b' = (s'_0, , s'_N) = stall(e_g(s_i)) \equiv_\tau b$. Since $s'_j = s_j, b' \notin \mathcal{B}^c_{mck}(i,j)$ iff inequality 3.4 does not hold for some $k \in \mathbb{N}$. In fact b' satisfies the other two conditions of Definition 3.5.4 because $b' \equiv_\tau b$ and to insert a stalling event on s_i (while s_j remains the same) cannot violate inequality 3.5. Now, suppose first that b' satisfies inequality 3.4 for all $k \in \mathbb{N}$: then, there exists at least a behavior that belongs to $\mathcal{MPE}[stall(e_g(s_i))] \cap \mathcal{B}^c_{mck}(i,j)$ and this behavior is b' because $\forall g \forall i, stall(e_g(s_i)) \in \mathcal{MPE}[stall(e_g(s_i))]$. A more interesting case is when inequality 3.4 does not hold: in this case $b' \notin \mathcal{B}^c_{mck}(i,j)$. Then, consider a behavior $b'' = (s''_1, \ldots, s''_N)$ s.t. $\forall n \in [1,N], (n \neq i \text{ and } n \neq j), (s''_n = s'_n)$, while s''_j is obtained from s'_j by inserting z number of stalls at $e_h(s'_j)$, where $e_h(s'_j) = e_h(s_j) = nextEvent(s_j, e_g(s_i))$ and

$$y = V_E(e_{g+1}(s_i)) - V_E(e_h(s_j))$$

$$z = \begin{cases} \left\lceil \frac{y}{T_j} \right\rceil, & \text{if } y > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Now according to the Definition 3.4.6 clearly, $b'' \in \mathcal{MPE}[stall(e_g(s_i))]$. In other words, insert stalls in s_j for z ticks, i.e., all ticks in s_j after h and before $g+1$ in s_i. This will delay s_j sufficiently to take care of stall in s_i.

Now in addition with inserting the above mentioned z stalls at $e_h(s'_j)$ if we insert any finite number of delays (less than K, see definition 3.4.6) at any event $e_a(s'_i)$ and $e_b(s'_j)$ s.t. $\forall a \forall b, a > g+1, b > h$ and $e_a(s'_i) \in \mathcal{E}_\iota(s'_i)$ and $e_b(s'_j) \in \mathcal{E}_\iota(s'_j)$, then we will get a behavior $b^\#$ s.t. $b^\# \in \mathcal{MPE}[stall(e_g(s_i))]$ (see definition 3.4.6). So now we need to prove that if we insert a stalling event at $e_g(s_i)$ then according to multiple clock procrastination effect we will get behavior $b^* \in \mathcal{MPE}[stall(e_g(s_i))]$ which satisfies the inequality 3.4 and 3.5 for all $k \in \mathbb{N}$.

Clearly, b'' satisfying inequalities 3.4 and 3.5 for all the events upto $e_{g+1}(s''_i)$ from the ordered set $realTimeOrdering(b'')$ [see definition 3.4.4]. But b'' may not satisfies the inequalities 3.4 and 3.5 for the events following $e_{g+1}(s''_i)$ in the above mentioned ordered set. Now let x be an informative event in the ordered set $realTimeOrdering(b'')$ such that all the events preceding x in the set are satisfying inequalities 3.4 and 3.5 (but not x). Note that if x is in s''_i then x can only violate inequality 3.5 and if x is in s''_j then it can only violate inequality 3.4. Now there may be two cases:

a) x is an event of s_i'' and it is not satisfying inequality 3.5.

Let $x = e_a(s_i'')$ where $a \in \mathbb{N}$ and $a > g+1$. Also let $e_b(s_j'')$ as $prevTimingEvent(s_j'', e_a(s_i''))$. Now x is not satisfying 3.5 means

$$\mathcal{F}_\iota[\sigma_{[t_0,t_a]}(s_i'')] - \mathcal{F}_\iota[\sigma_{[t_0,t_b]}(s_j'')] > c.$$

In particular:

$$\mathcal{F}_\iota[\sigma_{[t_0,t_a]}(s_i'')] - \mathcal{F}_\iota[\sigma_{[t_0,t_b]}(s_j'')] = c + 1.$$

To solve this problem we need to delay the events in s_i'' at a until an output of informative event is produced at s_j''. Inset stalls at $e_a(s_i'')$ until the $d = nextEvent(s_j'', e_b(s_j''))$ occurs. The number of stalls we need to insert at $e_a(s_i'')$ is

$$z' = \begin{cases} \left\lceil \frac{y'}{T_i} \right\rceil, & \text{if } y' > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Where,

$$y' = V_E(nextEvent(s_j'', e_b(s_j''))) - V_E(e_a(s_i'')).$$

b) x is an event of s_j'' and it is not satisfying inequality 3.4. Let $x = e_a(s_j'')$ where $a \in \mathbb{N}$ and $a > h + z$. Also let $e_b(s_i'')$ as $prevTimingEvent(s_i'', e_a(s_j''))$. Now x is not satisfying inequality 3.4 means

$$\mathcal{F}_\iota[\sigma_{[t_0,t_b]}(s_i'')] < \mathcal{F}_\iota[\sigma_{[t_0,t_a]}(s_j'')].$$

In particular:

$$\mathcal{F}_\iota[\sigma_{[t_0,t_b]}(s_i'')] + 1 = \mathcal{F}_\iota[\sigma_{[t_0,t_a]}(s_j'')].$$

To solve this problem we need to delay the event in s_j'' at a until an input of informative event take place at s_i''. i.e. insert stalls at $e_a(s_j'')$ until the event $d = nextEvent(s_i'', e_b(s_i''))$ occurs. Hence the number of stalls we need to insert at $e_a(s_j'')$ is

$$z' = \begin{cases} \left\lceil \frac{y'}{T_j} \right\rceil, & \text{if } y' > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Where,

$$y' = V_E(nextEvent(s_i'', e_b(s_i''))) - V_E(e_a(s_j'')).$$

Now let the behavior we get by inserting z' stalls in either cases mentioned above is $b^{1\#}$. Now $b^{1\#}$ will satisfy both inequality 3.4 and 3.5 for all the events up to an event y (including y) from the ordered set $realTimeOrdering(b^{1\#})$. Where the event y is an informative event in $b^{1\#}$ corresponding to the informative event x in b''.

So, it means that if there exist a behavior $b'' \in \mathcal{MPE}[stall(e_g(s_i))]$ and there exist an informative event x in b'' having the smallest real-time value among those informative events which are not satisfying either inequalities 3.4 or 3.5, then there must exist a behavior $b^{1\#} \in \mathcal{MPE}[stall(e_g(s_i))]$ which satisfies both inequality 3.4 and 3.5 up an informative event, which is the corresponding informative event of x in b''.

Now repeating the above procedure a finite number of times, we will get a behavior $b^{(N-1)\#} \in \mathcal{MPE}[stall(e_g(s_i))]$ which satisfies inequalities 3.4 and 3.5 up to the last-but-one informative event from the ordered set $realTimeOrdering(b^{(N-1)\#})$. It means that $b^{(N-1)\#}$ is not satisfying inequality 3.4 or 3.5 for the last informative event from the ordered set $realTimeOrdering(b^{(N-1)\#})$. If we repeat the above mentioned procedure for one more time then we will get a behavior $b^* \in \mathcal{MPE}[stall(e_g(s_i))]$ which will satisfy inequalities 3.4 and 3.5 completely.

Case 2: For all $v \in \mathbb{N}$, such that $e_v(s_j) \in \mathcal{E}_\iota(s_j)$, let $b' = (s'_0, \ldots, s'_N) = stall(e_v(s_j)) \equiv_\tau b$. Since $s'_i = s_i, b' \notin \mathcal{B}^c_{mck}(i,j)$ iff inequality 3.5 does not hold for some $k \in \mathbb{N}$. In fact b' satisfies the other two conditions of Definition 3.5.4 because $b' \equiv_\tau b$ and to insert a stalling event on s_j (while s_i remains the same) cannot violate inequality 3.4. Now, suppose first the b' satisfies inequality 3.5 for all $k \in \mathbb{N}$: then, there exists at least a behavior that belongs to $\mathcal{MPE}[stall(e_v(s_j))] \cap \mathcal{B}^c_{mck}(i,j)$ and this behavior is b' because $\forall v \forall j, stall(e_v(s_j)) \in \mathcal{MPE}[stall(e_v(s_j))]$. A more interesting case is when inequality 3.5 does not hold: in this case $b' \notin \mathcal{B}^c_{mck}(i,j)$. Then, consider a behavior $b'' = (s''_1, \ldots, s''_N)$ s.t. $\forall n \in [1,N], (n \neq j$ and $n \neq i), (s''_n = s'_n)$, while s''_i is obtained from s'_i by inserting z number of stalls at $e_h(s'_i)$, where $e_h(s'_i) = e_h(s_i) = nextEvent(s_i, e_v(s_j))$ and

$$y = V_E(e_{v+1}(s_j)) - V_E(e_h(s_i)).$$

$$z = \begin{cases} \left\lceil \frac{y}{T_i} \right\rceil, & \text{if } y > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Now according to the Definition 3.4.6 clearly, $b'' \in \mathcal{MPE}[stall(e_v(s_j))]$. In other words, insert stalls in s_i for z ticks, i.e., all ticks in s_i after h and before $v+1$ in s_j. This will delay s_i sufficiently to take care of stall in s_j.

Now, in addition with inserting the above mentioned z stalls at $e_h(s_i')$ if we insert any finite number of stalls (less than K, see Definition 3.4.6) at any event $e_a(s_i')$ and $e_b(s_j')$ s.t. $\forall a \forall b, a > h, b > v+1$ and $e_a(s_i') \in \mathcal{E}_\iota(s_i')$ and $e_b(s_j') \in \mathcal{E}_\iota(s_j')$, then we will get a behavior $b^{\#}$ s.t. $b^{\#} \in \mathcal{MPE}[stall(e_v(s_j))]$ (see Definition 3.4.6). So now we need to prove that if we insert a stalling event at $e_v(s_j)$ then according to multiple clock procrastination effect we will get a behavior $b^* \in \mathcal{MPE}[stall(e_v(s_j))]$, which satisfies the inequality 3.4 and 3.5 for all $k \in \mathbb{N}$.

Clearly, b'' satisfies inequalities 3.4 and 3.5 for all the events upto $e_{v+1}(s_j'')$ from the ordered set $realTimeOrdering(b'')$. But b'' may not satisfies the inequalities 3.4 and 3.5 for the events following $e_{v+1}(s_j'')$ in the above mentioned ordered set. Now let x be an informative event in the ordered set $realTimeOrdering(b'')$ such that all the events preceding x in the set are satisfying inequalities 3.4 and 3.5 (but not x). Note that if x is in s_i'' then x can only violate inequality 3.5 and if x is in s_j'' then it can only violate inequality 3.4. Now there may be two cases:

a) x is an event of s_i'' and it is not satisfying inequality 3.5. Let $x = e_a(s_i'')$ where $a \in \mathbb{N}$ and $a > h + z$. Also let $e_b(s_j'')$ as $prevTimingEvent(s_j'', e_a(s_i''))$. Now x is not satisfying 3.5 means:

$$\mathcal{F}_\iota[\sigma_{[t_0,t_a]}(s_i'')] - \mathcal{F}_\iota[\sigma_{[t_0,t_b]}(s_j'')] > c.$$

In particular:

$$\mathcal{F}_\iota[\sigma_{[t_0,t_a]}(s_i'')] - \mathcal{F}_\iota[\sigma_{[t_0,t_b]}(s_j'')] = c + 1.$$

To solve this problem we need to delay the events in s_i'' at a until an output of informative events produced at s_j''. i.e. inset stalls at $e_a(s_i'')$ until the $d = nextEvent(s_j'', e_b(s_j''))$ occurs. The number of stalls we need to insert at $e_a(s_i'')$ is

$$z' = \begin{cases} \left\lceil \frac{y'}{T_i} \right\rceil, & \text{if } y' > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Where,

$$y' = V_E(nextEvent(s_j'', e_b(s_j''))) - V_E(e_a(s_i'')).$$

b) x is an event of s_j'' and it is not satisfying inequality 3.4. Let $x = e_a(s_j'')$ where $a \in \mathbb{N}$ and $a > v + 1$. Also let $e_b(s_i'')$ as $prevTimingEvent(s_i'', e_a(s_j''))$. Now x is not satisfying inequality 3.4 means

$$\mathcal{F}_\iota[\sigma_{[t_0,t_b]}(s_i'')] < \mathcal{F}_\iota[\sigma_{[t_0,t_a]}(s_j'')]$$

In particular:

$$\mathcal{F}_\iota[\sigma_{[t_0,t_b]}(s_i'')] + 1 = \mathcal{F}_\iota[\sigma_{[t_0,t_a]}(s_j'')]$$

To solve this problem we need to delay the event in s_j'' at a until an input of informative event produces at s_i''. i.e. insert stalls at $e_a(s_j'')$ until the event $d = nextEvent(s_i'', e_b(s_i''))$ occurs. Hence the number of stalls we need to insert at $e_a(s_j'')$ is

$$z' = \begin{cases} \left\lceil \frac{y'}{T_j} \right\rceil, & \text{if } y' > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Where,

$$y' = V_E(nextEvent(s_i'', e_b(s_i''))) - V_E(e_a(s_j'')).$$

Now let the behavior we get by inserting z' stalls in either cases mentioned above is $b^{1\#}$. Now $b^{1\#}$ will satisfy both inequality 3.4 and 3.5 for all the events up to an event y (including y) from the ordered set $realTimeOrdering(b^{1\#})$. Where event y is an informative event in $b^{1\#}$ corresponding to the informative event x in b''.

So, it means that if there exist a behavior $b'' \in \mathcal{MPE}[stall(e_v(s_j))]$ and there exist an informative event x in b'' having the smallest real-time value among those informative events which are not satisfying either inequalities 3.4 or 3.5, then there must exist a behavior $b^{1\#} \in \mathcal{MPE}[stall(e_v(s_j))]$ which satisfies both inequality 3.4 and 3.5 up to an informative event, which is the corresponding informative event of x in b''.

Now repeating the above procedure a finite number of times, we will get a behavior $b^{(N-1)\#} \in \mathcal{MPE}[stall(e_v(s_j))]$ which satisfies inequalities 3.4 and 3.5 up to the last-but-one informative event from the ordered set $realTimeOrdering(b^{(N-1)\#})$. It means that $b^{(N-1)\#}$ is not satisfying inequality 3.4 or 3.5 for the last informative event from the ordered set $realTimeOrdering(b^{(N-1)\#})$. If we repeat the above mentioned procedure for one more time then we will get a behavior $b^* \in \mathcal{MPE}[stall(e_v(s_j))]$

which will satisfy inequalities 3.4 and 3.5 completely.

Case 3: Finally $\forall r \in ([1,N]/i,j)$. Let $b' = (s'_1,\ldots,s'_N) = stall(e_h(s_r)) \equiv_\tau b$, where for all $h \in \mathbb{N}, e_h(s_r) \in \mathcal{E}_\iota(s_r)$. Then trivially $b' \in \mathcal{MPE}[stall(e_h(s_r))] \cap \mathcal{B}^c_{mck}(i,j)$.

In conclusion, by combining all three cases, we have that $\forall b = (s_1,\ldots,s_N), \in \mathcal{B}^c_{mck}(i,j), \forall r \in [1,N], \forall e_k(s_r) \in \mathcal{E}_\iota(s_r), (\mathcal{MPE}[stall(e_k(s_r))] \cap \mathcal{B}^c_{mck}(i,j) \neq \emptyset)$.

Thus $\mathcal{B}^c_{mck}(i,j)$ is a multiple clock patient process. □

3.5.4 Multiple clock relay station

Lemma IV.3 of [2] proves that the single clock buffer $\mathcal{B}^2_{1,1}(i,j)$, is the minimum capacity buffer which is able to transfer one informative event per timestamp. The same Lemma is given here for completeness:

Lemma 3.5.4. *The single clock buffer $\mathcal{B}^2_{1,1}(i,j)$ is the minimum capacity buffer having forward and backward latencies equal to one, s.t. for all K, closed intervals of $\mathbb{N}$*

$$\begin{aligned}\exists b^K = (s^K_1,\ldots,s^K_N) \in \mathcal{B}^2_{1,1}(i,j) \wedge \forall k \in K,\\ (e_k(s^K_i) \in \mathcal{E}_\iota(s^K_i) \wedge e_k(s^K_j) \in \mathcal{E}_\iota(s^K_j)).\end{aligned} \tag{3.6}$$

□

But in case of multiple clock buffer, there is no minimum capacity for which it can receive and send informative events at every timestamp (clock tick). The following two cases explain why Lemma 3.5.4 is not valid for multiple clock buffers (even if we increase the capacity of the above single clock buffer).

1) Slower input faster output:- Consider $\mathcal{B}^c_{mck}(i,j)$ be the multiple clock buffer and let the clock ratio between s_i and s_j is $3:1$. Now, when the nth tick of s_i will occur, the number of ticks occurred in s_j up to this time, will be definitely greater than n. Therefore it is not possible to send informative event at every clock ticks of s_j, even if s_i is receiving informative event at its every clock ticks.

2) Faster input slower output:- Consider $\mathcal{B}^c_{mck}(i,j)$ be the multiple clock buffer and let the clock ratio between s_i and s_j is $1:3$. Now, when the nth tick of s_i will occur, the number of ticks occured in s_j up to this time, will be definitely less than n. Let the difference be k. Now this difference k will increase, if we calculate it between the time of $(n+1)$th ticks in s_i and the number of ticks in s_j up to this time. Similarly the difference will increase continuously. Therefore it is not possible to define a minimum capacity of the buffer $\mathcal{B}^c_{mck}(i,j)$ for which it can receive informative event at every clock tick of s_i, even if s_j sends informative event at its every clock tick.

The best thing possible is that the slower signal between s_i and s_j of $\mathcal{B}^c_{mck}(i,j)$ is able to send/receive informative events at its every clock tick. Clearly, this is possible only if the buffer capacity is greater than or equal to 2 (ignoring the synchronization issues for the FIFO discussed in [**?**]). Therefore we are using $\mathcal{B}^2_{mck}(i,j)$ as multiple clock relay station.

Definition 3.5.5. *The multiple clock buffer $\mathcal{B}^2_{mck}(i,j)$ is called the multiple clock relay station.* □

3.6 Multiple Clock Latency-Insensitive Design

Before going to explain the design methodology of multiple clock Latency Insensitive design the only thing remains to prove is that all the functional modules we are taking are also multiple clock patient processes. We only take those functional modules which can be converted to single clock patient process as explained in [2]. Therefore each functional module must belong to a single clock.

Lemma 3.6.1. *If any functional module can be converted to a single clock patient process then it can also be converted to a multiple clock patient process.*

Proof: The lemma can be easily proved because,

- Every single clock patient process is also a multiple clock patient process.
- Every single clock process which is not a patient process is also not a multiple clock patient process.

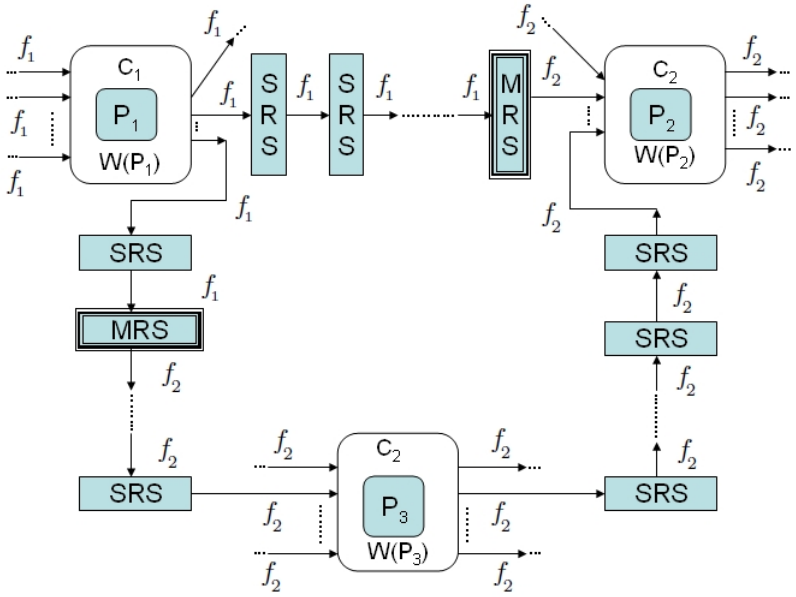

Figure 3.4: *An example of multiclock latency insensitive system.*

Therefore the procedure of converting a functional module to a multiple clock patient process is same as the procedure explained in [2], for converting it to a single clock patient process. □

3.6.1 Latency Insensitive Design Methodology for Multiple clock Domains

By putting all the ideas in the chapter together. We will list all the steps required to extend the latency insensitive design methodology to multiple clock domain.

1. Begin with a system of processes and channels.

2. Encapsulate each process with a wrapper as directed in the Latency Insensitive Design [2] in order to make the process patient and latency insensitive to the original one.

3. Insert the multiple clock relay station where communication is required between two domains not synchronous to each other. Then, create a latency insensitive

system by inserting single clock relay stations without affecting the correctness of the system.

In section 3.1, we said that the main aim of the multiple clock relay station is to eliminate synchronization errors which arise due to difference in clock frequency. Thus, we need to discuss the case where, even after eliminating the synchronization errors caused due to difference in frequencies, the latency induced due to the wire may generate errors while transmission. This situation can be handled very easily. By inserting a multiple clock relay station, we now have taken care of the synchronization and thus the traditional single clock relay stations can be introduced between synchronous signals as shown in the figure 3.4 and this will eliminate the synchronization error generated through the wire delay.

In figure 3.4, P_1, P_2 and P_3 are the single clock processes. Here P_1 belongs to clock C_1 and P_2, P_3 both belongs to clock C_2. $W(P_1), W(P_2)$ and $W(P_3)$ are the wrappers of processes P_1, P_2 and P_3 respectively. The wrappers are used here to make the processes as multiple clock patient. Note that the single clock patient processes are also multiple clock process. SRS means single clock relay station and MRS means multiple clock relay station. Each signal is labeled with its clock frequency. Signals belong to clock C_1 have frequency f_1 and signals belong to clock C_2 have frequency f_2. The whole system shown in figure 3.4 is a multiple clock patient process because each component (processes or relay stations) in the system are multiple clock patient processes (we proved them in this chapter). Hence their composition must also be a multiple clock patient process (we also proved this).

Thus, in this Chapter, we have given a similar methodology for orthogonalizing computation and communication even in the case of multiple clock domain and this will help us in building latency insensitive systems for multiple clocked systems. Also, the advantage of the approach followed is that it is compatible with the traditional single clock design and thus we can generate many number of new designs by composing traditional and current designs which has many advantages.

3.7 Conclusion

Latency issues in single clock domain systems have been handled in a previous work. We extended this work for connecting systems with multiple clock frequencies. The paradigm shift necessary to achieve this was demonstrated by means of examples where the original theory falls short in handling latency issues.

We defined a multiple clock LI buffer which can be used (i) in the chain of single clock relay stations; or (ii) simply as a converter, between two patient processing modules having rationally related frequencies, thus making the system LI. This buffer inputs data from one frequency domain and forwards it to another frequency domain; as well as it can tolerate back-pressure.

To handle latency issues created by input delays or back-pressure, the buffer has to insert stalls in various connecting signals. The number of stalls to be inserted depends on the ratio of the clock frequencies to be connected. The main observation being that the events in two sequences have co-relation every LCM(p_i, p_j) time units, where p_i and p_j are the clock periods of two connecting channels. The definition can be modified to take into account the setup and hold constraints of the storage elements, thus enabling implementability. It also guarantees that composing modules from the single clock and multiple clock LI design styles yield a multiple clock LI design.

Chapter 4

Related Works-2

4.1 Introduction

The term process algebra is used in different meanings. First of all, let me consider the word 'process'. It refers to behavior of a system. A system is anything showing behavior, in particular the execution of a software system, the actions of a machine or even the actions of a human being. Behavior is the total of events or actions that a system can perform, the order in which they can be executed and maybe other aspects of this execution such as timing or probabilities. Always, we describe certain aspects of behavior, disregarding other aspects, so we are considering an abstraction or idealization of the real behavior. Rather, we can say that we have an observation of behavior, and an action is the chosen unit of observation. Usually, the actions are thought to be discrete: occurrence is at some moment in time, and different actions are separated in time. This is why a process is sometimes also called a discrete event system. The word algebra denotes that we take an algebraic/axiomatic approach in talking about behavior. That is, we use the methods and techniques of universal algebra [**?**].

The simplest model of behavior is to see behavior as an input/output function. A value or input is given at the beginning of the process, and at some moment there is a value as outcome or output. This model was used to advantage as the simplest model of the behavior of a computer program in computer science, from the start of the subject in the middle of the twentieth century. It was instrumental in

the development of (finite state) automata theory. In automata theory, a process is modeled as an automaton. An automaton has a number of states and a number of transitions, going from one state to another state. A transition denotes the execution of an (elementary) action, the basic unit of behaviour. Besides, there is an initial state (sometimes, more than one) and a number of final states. A behavior is a run, i.e. a path from initial state to final state. Important from the beginning is when to consider two automata equal, expressed by a notion of equivalence. On automata, the basic notion of equivalence is language equivalence: a behavior is characterized by the set of executions from the initial state to a final state. An algebra that allows equational reasoning about automata is the algebra of regular expressions [**?**].

Later on, this model was found to be lacking in several situations. Basically, what is missing is the notion of interaction: during the execution from initial state to final state, a system may interact with another system. This is needed in order to describe parallel or distributed systems, or so-called reactive systems. When dealing with interacting systems, we say we are doing concurrency theory, so concurrency theory is the theory of interacting, parallel and/or distributed systems. When talking about process algebra, we usually consider it as an approach to concurrency theory, so a process algebra will usually (but not necessarily) have parallel composition as a basic operator.

Thus, we can say that process algebra is the study of the behavior of parallel or distributed systems by algebraic means. It offers means to describe or specify such systems, and thus it has means to talk about parallel composition. Besides this, it can usually also talk about alternative composition (choice) and sequential composition (sequencing). Moreover, we can reason about such systems using algebra, i.e. equational reasoning. By means of this equational reasoning, we can do verification, i.e. we can establish that a system satisfies a certain property.

In this chapter we discuss some important languages of process algebra that we have used in our work. Section 2 discuses the most popular language of process algebra called CCS [**?**]. Another version of process algebra called CSP [] is discuss in section 3. Section 4 to 6 describes some process algebras with timing constraints. Which are also called timed process algebra.

4.2 CCS:

The central person in the history of process algebra without a doubt is Robin Milner. A.J.R.G. Milner, born in 1934, developed his process theory CCS (Calculus of Communicating Systems) over the years 1973 to 1980, culminating in the publication of the book [**?**] in 1980.

We shall now introduce the language CCS. We begin by informally presenting the process constructions allowed in this language and their semantics in Section 4.2.1. We then proceed to put our developments on a more formal footing in Section **??**.

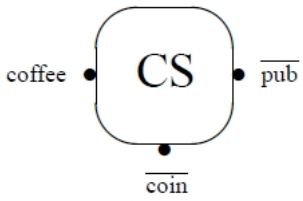

Figure 4.1: *The interface for process CS.*

4.2.1 Some CCS Process Constructions

It is useful to begin by thinking of a CCS process as a black box. This black box may have a name that identifies it, and has a p*rocess interface*. This interface describes the collection of *communication ports*, also referred to as *channels*, that the process may use to interact with other processes that reside in its environment, together with an indication of whether it uses these ports for inputting or outputting information. For example, the drawing in Figure 4.1 pictures the interface for a process whose name is CS (for Computer Scientist). This process may interact with its environment via three ports, or communication channels, namely $coffee$, $\overline{coin}$ and *pub*. The port $coffee$ is used for input, whereas the ports $\overline{coin}$ and $\overline{pub}$ are used by process CS for output. In general, given a port name a, we use a for output on port $\overline{a}$. We shall often refer to labels as coffee or coin as actions.

A description like the one given in Figure 4.1 only gives static information about a process. What we are most interested in is the behavior of the process being specified.

The behavior of a process is described by giving a "CCS program". The idea being that, as we shall see soon, the process constructions that are used in building the program allow us to describe both the structure of process and its behavior.

Let us begin by introducing the constructs of the language CCS by means of examples.

Example 4.1. The most basic process of all is the process **0** (read "nil"). This is the most boring process imaginable, as it performs no action whatsoever. The process **0** offers the prototypical example of a deadlocked behaviorone that cannot proceed any further in its computation.

Example 4.2. The most basic process constructor in CCS is action prefixing. Two example processes built using **0** and action prefixing are:

- **a match:** strike.**0** and,
- **a complex match:** take.strike.**0**.

Intuitively, a match is a process that dies when stricken (i.e., that becomes the process **0** after executing the action strike), and a complex match is one that needs to be taken before it can behave like a match.

More in general, the formation rule for action prefixing says that: If P is a process and a is a label, then $a.P$ is a process. The idea is that a label, like $strike$ or $\overline{pub}$, will denote an input or output action on a communication port, and that the process $a.P$ is one that begins by performing action a and behaves like P thereafter.

Example 4.3. Consider that of a simple coffee vending machine, which always gives a cup of coffee after inserting a coin and thereafter return to its initial state. The process will be as follows:

$$CM \stackrel{def}{=} coin.\overline{coffee}.CM. \tag{4.1}$$

This type of processes are called recursive process.

Example 4.4. The CCS constructs that we have presented so far would not allow us to describe the behavior of a vending machine that allows its paying customer to choose between tea and coffee, say. In order to allow for the description of processes whose behavior may follow different patterns of interaction with their environment,

CCS offers the *choice operator*, which is denoted $+$. For example, a vending machine offering either tea or coffee may be described as follows:

$$CTM \stackrel{def}{=} coin.(\overline{coffee}.CTM + \overline{tea}.CTM). \tag{4.2}$$

The idea here is that, after having input a coin, the process CTM is willing to deliver either coffee or tea, depending on its customers choice. In general, the formation rule for choice states that: If P and Q are processes, then so is $P + Q$. The process $P + Q$ is one that has the initial capabilities of both P and Q. However, choosing to perform initially an action from P will preempt the further execution of actions from Q, and vice versa.

Example 4.5. Assume that a scientist working in a academia needs a cup of coffee after each of his publication. The behavior of such an academic may be described by the CCS process as follows:

$$CS \stackrel{def}{=} \overline{pub}.\overline{coin}.coffee.CS. \tag{4.3}$$

As made explicit by the above description, a computer scientist is initially keen to produce a publication, but she needs coffee to produce her next publication. Coffee is only available through interaction with the departmental coffee machine CM.

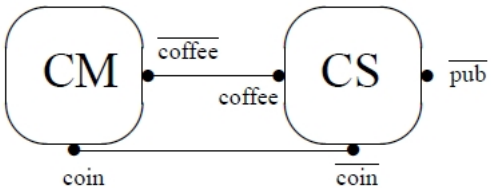

Figure 4.2: *The interface for process* $CM|CS$.

In order to describe systems consisting of two or more processes running in parallel, and possibly interacting with each other, CCS offers the *parallel composition operation* $|$. For example, the CCS expression $CM|CS$ describes a system consisting of two processesthe coffee machine CM and the computer scientist CSthat run in parallel one with the other. These two processes may communicate via the communication ports they share and use in complementary fashion, namely $coffee$ and $coin$. By

complementary, we mean that one of the processes uses the port for input and the other for output. Potential communications are represented in Figure 4.2 by the solid lines linking complementary ports. The port pub is instead used by the computer scientist to communicate with its research environment, or, more prosaically, with other processes that may be present in its environment and that are willing to input along that port. One important thing to note is that the link between complementary ports in Figure 4.2 denotes that it is possible for the computer scientist and the coffee machine to communicate in the parallel composition $CM|CS$. However, we do not require that they must communicate with one another. Both the computer scientist and the coffee machine could use their complementary ports to communicate with other reactive systems in their environment. For example, another computer scientist $CS0$ can use the coffee machine CM(See Figure 4.3.)

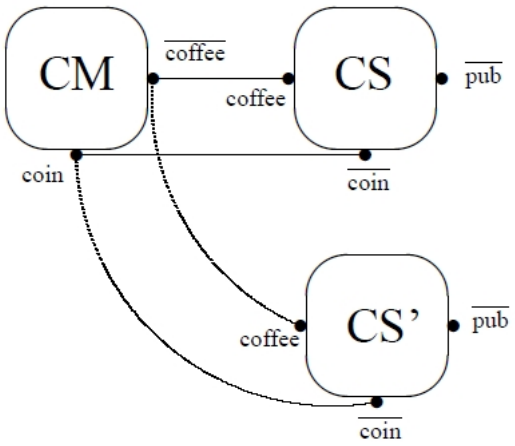

Figure 4.3: *The interface for process $CM|CS|CS'$.*

In general, given two CCS expressions P and Q, the process $P|Q$ describes a system in which

- P and Q may proceed independently and
- may communicate via complementary ports.

Example 4.6. Since academics like the computer scientist often live in a highly competitive environment, it may be fruitful for him to make the coffee machine

CM private to him, and therefore inaccessible to his competitors. To make this possible, the language CCS offers an operation called *restriction*, whose aim is to delimit the scope of channel names in much the same way as variables have scope in block structured programming languages. For instance, using the operations $/coin$ and $/coffee$, we may hide the $coin$ and $coffee$ ports from the environment of the processes CM and CS. Define

$$SmUni \stackrel{def}{=} (CM|CS)/coin/coffee. \tag{4.4}$$

As pictured in Figure 4.4, the restricted $coin$ and $coffee$ ports may now only be used for communication between the computer scientist and the coffee machine, and are not available for interaction with their environment. Their scope is restricted to the process $SmUni$. The only port of $SmUni$ that is visible to its environment, e.g., to the competing computer scientist $CS0$, is the one via which the computer scientist CS outputs his publications.

In general, the formation rule for restriction is:

> If P is a process and L is a set of port names, then P/L is a process.

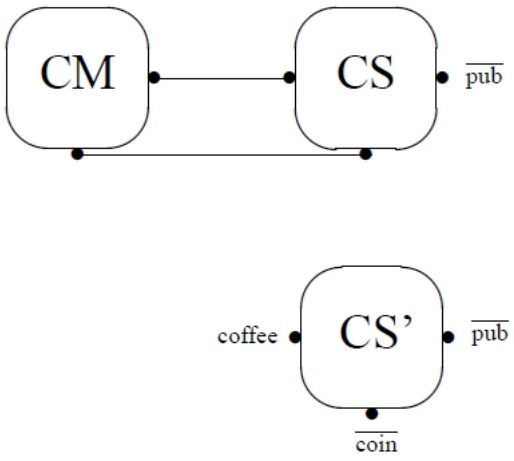

Figure 4.4: *The interface for processes $SmUni$ and $SmUni|CS'$.*

Example 4.7. Since a computer scientist cannot live on coffee alone, it is beneficial for him to have access to other types of vending machines offering, say, chocolate,

dried figs and crisps. The behavior of these machines may be easily specified by means of minor variations on 4.1. For instance, we may define the processes

$$CHM \stackrel{def}{=} coin.\overline{choc}.CHM \tag{4.5}$$

$$DFM \stackrel{def}{=} coin.\overline{figs}.DFM \tag{4.6}$$

$$CRM \stackrel{def}{=} coin.\overline{crisps}.CRM. \tag{4.7}$$

Note, however, that all of these vending machines follow a common behavioral pattern, and may be seen as specific instances of a generic vending machine that inputs a coin, dispenses an item and restarts, namely the process

$$VM \stackrel{def}{=} coin.\overline{item}.VM. \tag{4.8}$$

All of the aforementioned specific vending machines may be obtained as appropriate "renaming" of VM. For example,

$$CHM \stackrel{def}{=} VM[choc/item], \tag{4.9}$$

where $VM[choc/item]$ is a process that behaves like VM, but outputs chocolate whenever VM dispenses the generic item. In general,

> If P is a process and f is a function from labels to labels satisfying certain requirements, then $P[f]$ is a process.

By introducing the *relabeling operation*, we have completed our informal tour of the operations offered by the language CCS for the description of process behaviors. We shall now expand a little on the connection between CCS expressions and the automata describing their behavior. The presentation will again be informal, as we plan to highlight the main ideas underlying this connection rather than to focus immediately on the technicalities.

4.2.2 The Behavior of Processes

The key idea underlying the semantics of CCS is that a process passes through states during an execution; processes change their state by performing actions. For instance, the process CS defined in 4.3 can perform action $\overline{pub}$ and evolve into a process whose behavior is described by the CCS expression

$$CS1 \stackrel{def}{=} \overline{coin}.coffee.CS \tag{4.10}$$

in doing so. Process $CS1$ can then output a coin, thereby evolving into a process whose behavior is described by the CCS expression

$$CS2 \stackrel{def}{=} coffee.CS \tag{4.11}$$

Finally, this process can input coffee, and behave like our old CS all over again. Thus the processes $CS, CS1$ and $CS2$ are the only possible states of the computation of process CS. Note, furthermore, that there is really no conceptual difference between processes and their states! By performing an action, a process evolves to another process that describes what remains to be executed of the original one.

In CCS, processes change state by performing transitions, and these transitions are labeled by the action that caused them. An example state transition is

$$CS \stackrel{\overline{pub}}{\rightarrow} CS1, \tag{4.12}$$

which says that CS can perform action pub, and become CS1 in doing so.

In much the same way, we can make explicit the set of states of the coffee machine described in 4.1 by rewriting that equation thus:

$$CM \stackrel{def}{=} coin.CM1. \tag{4.13}$$

$$CM1 \stackrel{def}{=} \overline{coffee}.CM. \tag{4.14}$$

Note that the computer scientist is willing to output a coin in state $CS1$, as witnessed by the transition

$$CS1 \stackrel{\overline{coin}}{\rightarrow} CS2, \tag{4.15}$$

and the coffee machine is willing to accept that coin in its initial state, because of the transition

$$CM \stackrel{coin}{\rightarrow} CM1, \tag{4.16}$$

Therefore, when put in parallel with one another, these two processes may communicate and change state simultaneously. The result of the communication should be described as a state transition of the form

$$CM|CS1 \stackrel{?}{\rightarrow} CM1|CS2. \tag{4.17}$$

However, we are now faced with an important design decisionnamely, we should decide what label to use in place of the "?" labeling the above transition. Should

we decide to use a standard label denoting input or output on some port, then a third process might be able to synchronize further with the coffee machine and the computer scientist, leading to multi-way synchronization. The choice made by Milner in his design of CCS is different. In CCS, communication is via handshake, and leads to a state transition that is unobservable, in the sense that it cannot synchronize further. This state transition is labelled by a new label τ . So the above transition is indicated by

$$CM|CS1 \xrightarrow{\tau} CM1|CS2. \tag{4.18}$$

4.2.3 CCS, Formally

Having introduced CCS by example, we now proceed to present formal definitions for its syntax and semantics. We have already indicated in our examples how the operational semantics for CCS can be given in terms of automatawhich we have called Labeled Transition Systems as customary in concurrency theory. These we now proceed to define, for the sake of clarity.

Definition 4.2.1. *A labelled transition system (LTS) is a triple* $(Proc, Act, \left\{\xrightarrow{a} |a \in Act\right\})$, *where*

- $Proc$ *is a set of states, ranged over by s;*
- Act *is a set of actions, ranged over by a;*
- $\xrightarrow{a} \subseteq Proc \times Proc$ *is a transition relation, for every* $a \in Act$. *As usual, we shall use the more suggestive notation* $s \xrightarrow{a} s'$ *in lieu of* $(s, s') \in \xrightarrow{a}$, *and write* s *refuses* a *iff* $s \xrightarrow{a} s'$ *for no state* s'.

The first step in our formal developments is to offer the formal syntax for the language CCS. Since the set of ports plays a crucial role in the definition of CCS processes, we begin by assuming a countably infinite collection $\mathcal{A}$ of (*channel*) *names*. ("Countably infinite" means that we have as many names as there are natural numbers.) The set

$$\overline{\mathcal{A}} = \{\overline{a}|a \in \mathcal{A}\}$$

is the set of *complementary names* (or *co-names* for short). In our informal introduction to the language, we have interpreted names as input actions and co-names as output actions. We let

$$\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$$

be the set of labels, and

$$Act = \mathcal{L} \cup \{\tau\}$$

be the set of *actions*. In our formal developments, we shall use a, b to range over L, but, as we have already done in the previous section, we shall often use more suggestive names for channels in applications and examples. By convention, we assume that $\overline{a} = a$ for each label a. (This also makes sense intuitively because the complement of output is input.) We also assume a given countably infinite collection $\mathcal{K}$ of *process names* (or *constants*). (This ensures that we never run out of names for processes.)

Definition 4.2.2. *The collection P of CCS expressions is given by the following grammar:*

$$P, Q ::= K \quad | \quad \alpha.P \quad | \quad \sum_{i \in I} P_i \quad | \quad P|Q \quad | \quad P[f] \quad | \quad P\backslash L,$$

where,

- K *is process name in* $\mathcal{K}$*;*
- α *is an action in* Act*;*
- I *is an index set;*
- $f : Act \to Act$ *is a relabeling function satisfying the following constraints:*
  - $f(\tau) = \tau$ *and*
  - $f(\overline{a}) = \overline{f(a)}$ *for each label* a*;*
- *L is a set of labels.*

We write **0** for an empty sum of processes, i.e.,

$$\sum_{i \in \emptyset} P_i,$$

and $P_1 + P_2$ for a sum of two processes, i.e.,

$$P_1 + P_2 = \sum_{i \in \{1,2\}} P_i.$$

Moreover, we assume that the behaviour of each process constant is given by a defining equation

$$K \stackrel{def}{=} P.$$

As it was already made clear by the previous informal discussion, the constant K may appear in P.

To formally capture our understanding of the semantics of the language CCS, we therefore introduce the collection of SOS rules in the following Table. A transition $P \stackrel{\alpha}{\rightarrow} Q$ holds for CCS expressions P, Q if, and only if, it can be proven using these rules.

$$\frac{P \stackrel{\alpha}{\rightarrow} P'}{K \stackrel{\alpha}{\rightarrow} P'} \; K \stackrel{def}{=} P \qquad \frac{}{\alpha.P \stackrel{\alpha}{\rightarrow} P} \qquad \frac{P_j \stackrel{\alpha}{\rightarrow} P'_j}{\sum_{i \in I} P_i \stackrel{\alpha}{\rightarrow} P'_j} \; j \in I$$

$$\frac{P \stackrel{\alpha}{\rightarrow} P'}{P|Q \stackrel{\alpha}{\rightarrow} P'|Q} \qquad \frac{Q \stackrel{\alpha}{\rightarrow} Q'}{P|Q \stackrel{\alpha}{\rightarrow} P|Q'} \qquad \frac{P \stackrel{\alpha}{\rightarrow} P' \quad Q \stackrel{\overline{\alpha}}{\rightarrow} Q'}{P|Q \stackrel{\tau}{\rightarrow} P'|Q'}$$

$$\frac{P \stackrel{\alpha}{\rightarrow} P'}{P[f] \stackrel{f(\alpha)}{\rightarrow} P'[f]} \qquad \frac{P \stackrel{\alpha}{\rightarrow} P'}{P \backslash L \stackrel{\alpha}{\rightarrow} P' \backslash L} \; \alpha, \overline{\alpha} \notin L$$

A rule like

$$\frac{}{\alpha.P \stackrel{\alpha}{\rightarrow} P}$$

is an *axiom*, as it has no *premises*that is, it has no transition above the solid line. This means that proving that a process of the form $\alpha.P$ affords the transition $\alpha.P \stackrel{\alpha}{\rightarrow} P$ (the *conclusion* of the rule) can be done without establishing any further sub-goal. Therefore each process of the form $\alpha.P$ affords the transition $\alpha.P \stackrel{\alpha}{\rightarrow} P$. As an example, we have that the following transition:

$$\overline{pub}.\overline{coin}.coffee.CS \stackrel{\overline{pub}}{\rightarrow} \overline{coin}.coffee.CS$$

is provable using the above rule for action prefixing.

An example of a rule with a side condition is that for restriction, namely

$$\frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\alpha, \overline{\alpha} \notin L$$

This rule states that every transition of a term P determines a transition of the expression $P\backslash L$, provided that neither the action producing the transition nor its complement are in L.

4.2.4 Value Passing CCS

So far, we have only introduced the so-called pure CCSthat is, the fragment of CCS where communication is pure synchronization and involves no exchange of data. In many applications, however, processes exchange data when they communicate. We shall now introduce the new features in this language, and their operational semantics, by means of examples. In what follows, we shall assume for simplicity that the only data type is the set of non-negative integers.

Example 4.8. Assume that we wish to define a one-place buffer B which has the following behavior:

- If B is empty, then it is only willing to input one datum along a channel called 'in'. The received datum is stored for further output.
- If B is full, then it is only willing to output the successor of the value it stores, and empties itself in doing so.

This behavior of B can be modeled in value passing CCS thus:

$$B \stackrel{def}{=} in(x).B(x)$$

$$B(x) \stackrel{def}{=} \overline{out}(x+1).B$$

Note that the input prefix 'in' now carries a parameter that is a variablein this case $x$whose scope is the process that is prefixed by the input actionin this example, $B(x)$.

4.3 CSP

A very important development of process algebra is CSP [**?**], [**?**]. Them main concepts of CCS and CSP are same they made for modeling the interactive systems. Hence we just discuss the basic syntax of CSP. Actually CSP is a huge language and it is not possible to discuss everything here. We are just describing the basic syntaxes of CSP. The deltails can be found in [**?**] and [**?**].

4.3.1 Process

Definition 4.3.1. *The set of names of events which are considered relevant for a particular description of an object is called its alphabet. The alphabet is a permanent predefined property of an object. It is logically impossible for an object to engage in an event outside its alphabet. Alphabets of process P is denoted by αP.*

4.3.1.1 $STOP$ process:

The $STOP$ is a process which never engages in any event. In other words we can say that $STOP$ is a deadlock process.

4.3.1.2 Prefix

It is most important and basic process in CSP. Let x be an event and let P be a process. Then

$$x \rightarrow P \quad \text{pronounced as } x \text{ then } P.$$

describes an object which first engages in the event x and then behaves exactly as described by P.

4.3.1.3 Recursion

The prefix notation can be used to describe the entire behaviour of a process that eventually stops. But it would be extremely tedious to write out the full behavior of a repetitive process for its maximum design life; so we need a method of describing repetitive behaviours by much shorter notations. Consider the simplest possible everlasting object, a clock which never does anything but tick.

$$\alpha Clock = \ tick\}$$

Consider next an object that behaves exactly like the clock, except that it first emits a single tick

$$(tick \rightarrow Clock)$$

The behaviour of this object is indistinguishable from that of the original clock. This reasoning leads to formulation of the equation

$$Clock = tick \rightarrow Clock$$

4.3.1.4 Choice

By means of prefixing and recursion it is possible to describe objects with a single possible stream of behaviour. However, many objects allow their behaviour to be influenced by interaction with the environment within which they are placed. If x and y are distinct events

$$(x \rightarrow P|y \rightarrow Q)$$

describes an object which initially engages in either of the events x or y. After the first event has occurred, the subsequent behaviour of the object is described by P if the first event was x, or by Q if the first event was y. Since x and y are different events, the choice between P and Q is determined by the first event that actually occurs.

4.3.1.5 Trace

A trace of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. Imagine there is an observer with a notebook who watches the process and writes down the name of each event as it occurs. We can validly ignore the possibility that two events occur simultaneously; for if they did, the observer would still have to record one of them first and then the other, and the order in which he records them would not matter.

A trace will be denoted as a sequence of symbols, separated by commas and enclosed in angular brackets

$$\langle x, y \rangle \, consistsoftwoevents$$

x$followedby$y.

4.3.1.6 Event Renamig

When a process's pattern of communication has been described in terms of particular events, it is possible to obtain a new process by renaimimng those events. Its executions are essentially those of the original process but where the events are renamed. This allows the reuse of particular descriptions of process behaviour without the need to rewrite the process in full and replacing all the original event names with the new ones.

The process $f(P)$ is able to perform an event $f(a)$ precisely when the process P could perform the corresponding event a. Furthermore, $f(P)$ can perform internal events whenever P can. This behavior is captured by the following transition rules:

$$\frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')}$$
$$\frac{P \xrightarrow{\tau} P'}{f(P) \xrightarrow{\tau} f(P')}$$

Where a is an external event and τ is an internal event.

4.3.2 Concurrency

4.3.2.1 Interaction

When two processes are brought together to evolve concurrently, the usual intention is that they will interact with each other. These interactions may be regarded as events that require simultaneous participation of both the processes involved. For the time being, let us confine attention to such events, and ignore all others. Thus we will assume that the alphabets of the two processes are the same. Consequently, each event that actually occurs must be a possible event in the independent behaviour of each process separately.

If P and Q are processes with the same alphabet, we introduce the notation

$$P||Q$$

to denote the process which behaves like the system composed of processes P and Q interacting in lock-step synchronization as described above.

4.3.2.2 Concurrency

The operator described in the previous section can be generalised to the case when its operands P and Q have different alphabets

$$\alpha P \neq \alpha Q$$

When such processes are assembled to run concurrently, events that are in both their alphabets (as explained in the previous section) require simultaneous participation of both P and Q. However, events in the alphabet of P but not in the alphabet of Q are of no concern to Q, which is physically incapable of controlling or even of noticing them. Such events may occur independently of Q whenever P engages in them. Similarly, Q may engage alone in events which are in the alphabet of Q but not of P. Thus the set of all events that are logically possible for the system is simply the union of the alphabets of the component processes

$$\alpha(P||Q) = \alpha P \cup \alpha Q.$$

4.3.2.3 Pictures

A process P with alphabet $\{a, b, c\}$ is pictured as a black-box labeled P, from which emerge a number of lines, each labeled with a different event from its alphabet (Figure 4.5(a)). Similarly, Q with its alphabet $\{b, c, d\}$ may be pictured as in Figure 4.5(b). When these two processes are put together to evolve concurrently, the resulting system

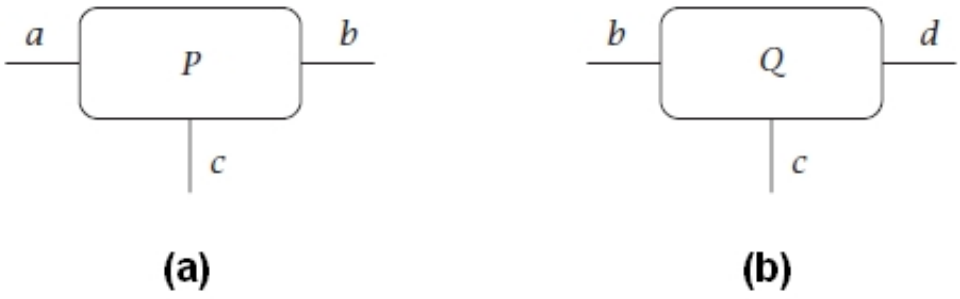

Figure 4.5: *The black box diagram for P and Q.*

may be pictured as a network in which similarly labeled lines are connected, but lines labeled by events in the alphabet of only one process are left free (Figure 4.6).

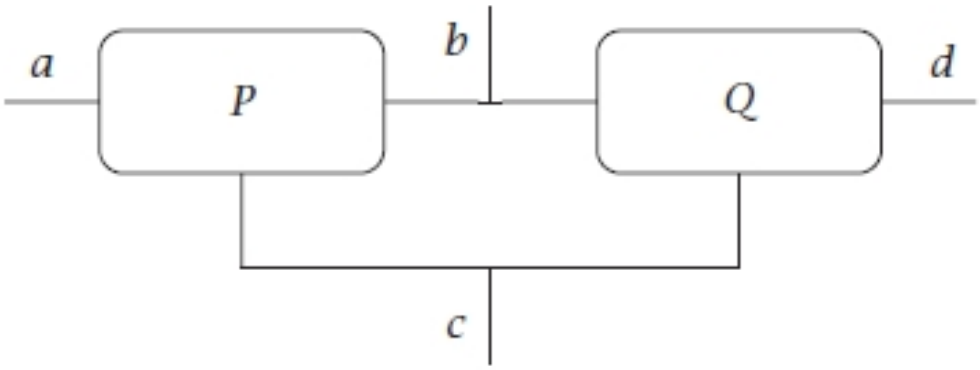

Figure 4.6: *The black box diagram for P and Q.*

4.3.2.4 Process Labeling

Change of symbol is particularly useful in constructing groups of similar processes which operate concurrently in providing identical services to their common environment, but which do not interact with each other in any way at all. This means that they must all have different and mutually disjoint alphabets. To achieve this, each process is labeled by a different name; and each event of a labeled process is also labeled by its name. A labelled event is a pair $l.x$ where l is a label, and x is the symbol standing for the event. A process P labelled by l is denoted by

$$l : P$$

It engages in the event $l.x$ whenever P would have engaged in x. The function required to define $l : P$ is

$$f_l(x) = l.x \quad \text{for all } x \text{ in } \alpha P. and the definition of labeling is, l : P = f_l(P)$$

4.3.3 Nondeterminism

The choice operator | is used to define a process which exhibits a range of possible behaviors; and the concurrency operator || permits some other process to make a selection between the alternatives offered.

Such processes are called deterministic, because whenever there is more than one event possible, the choice between them is determined externally by the environment of the process. It is determined either in the sense that the environment can actually

make the choice, or in the weaker sense that the environment can observe which choice has been made at the very moment of the choice.

Sometimes a process has a range of possible behaviours, but the environment of the process does not have any ability to influence of even observe the selection between the alternatives. For example, a different change-giving machine may give change in either of the combinations described above; but the choice between them cannot be controlled or even predicted by its user. The choice is made, as it were internally, by the machine itself, in an arbitrary or nondeterministic fashion. The environment cannot control the choice of even observe it; it cannot find out exactly when the choice was made, although it may later infer which choice was made from the subsequent behaviour of the process.

The nondeterminism is useful for maintaining a high level of abstraction in descriptions of the behaviour of physical systems and machines.

4.3.3.1 General choice

General choice means an operation

$$(P \Box Q),$$

for which the environment can control which of P and Q will be selected, provided that this control is exercised on the very first action. If this action is not a possible first action of P, then Q will be selected; but if Q cannot engage initially in the action, P will be selected. If, however, the first action is possible for both P and Q, then the choice between them is nondeterministic.

4.3.3.2 Concealment

In general, the alphabet of a process contains just those events which are considered to be relevant, and whose occurrence requires simultaneous participation of an environment. In describing the internal behaviour of a mechanism, we often need to consider events representing internal transitions of that mechanism. Such events may denote the interactions and communications between concurrently acting components from which the mechanism has been constructed. After construction of the mechanism, it is possible to conceal the structure of its components; and also to conceal all occurrences of actions internal to the mechanism. In fact, we want these actions to

occur automatically and instantaneously as soon as they can, without being observed or controlled by the environment of the process. If C is a finite set of events to be concealed in this way, then

$$P \backslash C$$

is a process which behaves like P, except that each occurrence of any event in C is concealed.

4.3.3.3 Interleaving

The || operator was defined in section **??** in such a way that actions in the alphabet of both operands require simultaneous participation of them both, whereas the remaining actions of the system occur in an arbitrary interleaving. Using this operator, it is possible to combine interacting processes with differing alphabets into systems exhibiting concurrent activity, but without introducing nondeterminism.

However, it is sometimes useful to join processes with the same alphabet to operate concurrently without directly interacting or synchronizing with each other. In this case, each action of the system is an action of exactly one of the processes If one of the processes cannot engage in the action, then it must have been the other one; but if both processes could have engaged in the same action, the choice between them is nondeterministic. This form of combination is denoted

$$P|||Q \quad P \text{ interleave } Q.$$

4.3.4 Communication

A communication is an event that is described by a pair $c.v$ where c is the name of the channel on which the communication takes place and v is the value of the message which passes.

4.3.4.1 Input and output

Let v be any member of $\alpha c(P)$. A process which first outputs v on the channel c and then behaves like P is defined

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

The only event in which this process is initially prepared to engage is the communication event $c.v$.

A process which is initially prepared to input any value x communicable on the channel c, and then behave like $P(x)$, is defined

$$c?x \rightarrow P(x)$$

4.3.4.2 Picture

When drawing a black box diagram of a process, the channels are drawn as arrows in the appropriate direction, and labeled with the name of the channel as shown in the figure 4.5.

Let P and Q be processes, and let c be an output channel of P and an input channel of Q. When P and Q are composed concurrently in the system $(P||Q)$, communication will occur on channel c on each occasion that P outputs a message and Q simultaneously inputs that message. An outputting process specifies a unique value for the message, whereas the inputting process is prepared to accept any communicable value. Thus the event that will actually occur is the communication $c.v$, where v is the value specified by the outputting process. This requires the obvious constraint that the channel c must have the same alphabet at both ends.

4.3.4.3 Communications

Let P and Q be processes, and let c be a channel used for output by P and for input by Q. Thus the set containing all communication events of the form $c.v$ is within the intersection of the alphabet of P with the alphabet of Q. When these processes are composed concurrently in the system $(P||Q)$, a communication $c.v$ can occur only when both processes engage simultaneously in that event, i.e., whenever P outputs a value v on the channel c, and Q simultaneously inputs the same value. An inputting process is prepared to accept any communicable value, so it is the outputting process that determines which actual message value is transmitted on each occasion.

4.3.5 Sequential Processes

The process $STOP$ is defined as one that never engages in any action. It is not a useful process, and probably results from a deadlock or other design error, rather than

a deliberate choice of the designer. However, there is one good reason why a process should do nothing more, namely that it has already accomplished everything that it was designed to do. Such a process is said to terminate successfully. In order to distinguish between this and STOP, it is convenient to regard successful termination as a special event, denoted by the symbol $\surd$ (pronounced "success"). A sequential process is defined as one which has $\surd$ in its alphabet; and naturally this can only be the last event in which it engages.

4.3.5.1 The Process $SKIP$

$SKIP$ is defined as a process which does nothing but terminate successfully. For example, a vending machine that is intended to serve only one customer with chocolate or toffee and then terminate successfully is,

$$VMONE = (coin \rightarrow (choc \rightarrow SKIP | toffee \rightarrow SKIP)).$$

4.3.5.2 Sequential Composition

If P and Q are sequential processes with the same alphabet, their sequential composition

$$P; Q$$

is a process which first behaves like P; but when P terminates successfully, $(P; Q)$ continues by behaving as Q. If P never terminates successfully, neither does $(P; Q)$.

4.4 Timed Process Algebra (TPL)

Time is often an important acpect of the deccription of many concurrent systems but it is not directly represented in any of the standard process algebra like CCS [] and CSP []. The authors of [**?**] proposes a Timed Process Algebra called TPL, which is actually an extension of CCS defined in section **??**.

The idea is to introduce into a standard process algebra, CCS, a σ action. The execution of this σ action by a process indicates that it is idling or doing nothing until the next clock cycle. This action will share many of the properties of the standard actions of CCS but because it represent the passage of time it will be distinguished from the standard actions by certain of its properties. In TPL, this

action is *deterministic* in the sense that a process can only reach at most one new state by performing σ. This is a reflection of the assumption that the passage of time is *deterministic*.

The main properties of TPL are as follows:

- *Discrete Time:* In TPL time proceeds in discrete steps represented by occurrence of an action σ.
- *Time Determinism:* TPL assumes the passage of time as *deterministic*.
- *Instantaneous Actions:* Time is not associated directly with communication actions but occurs independently.
- *Patience:* Process will wait indefinitely until they can communicate.
- *Maximal Progress:* Processes communicate as soon as a possibility of communication arise.

4.4.1 Syntax and semantics of TPL

In this section we discuss the syntax of TPL briefly. The abstract syntax of TPL ig given by the BNF definition

$$\begin{aligned} t ::= \quad & nill|\Omega|x|\sigma.t|\ \lfloor t \rfloor\ (t)|a.t|\tau.t| \\ & t+t|(t|t)|t[S]|t\backslash a|recx.t. \end{aligned}$$

where a ranges over Act, a set of actions not containing the distinguished actions τ and σ. Now we are giving some explanation about each of this operatots:

- $nill$ **:** This is a process which is terminated or deadlocked. It can perform no action but will allow the passage of time.
- Ω **:** This process represents incomplete information or divergence.
- $a.$ **:** The process $a.P$ can perform an action a and is so doing evolve into the process P. Because in TPL all processes are patient, $a.P$ will be able to ideal indefinitely until the a action is requested by the environment.

- τ **:** This is the silent or internal action in TPL. Since TPL using the assumption of maximal progress, $\tau.P$ will not be able to ideal in any environment. The τ action represent the internal communication.
- σ **:** The of time is modeled in TPL as the occurrence of a σ action. It represents a very abstruct notion of time, but it can be intuitively thought of as the click of a clock which measures the passage of time for the system.
- $+$ **:** This is borrowed from CCS and its meaning is almost same as CCS. The difference comes with the action σ. If the two processes are just idling before the environment request one of them, the choice between them will not be made by the passage of time alone. That is to say, $+$ is not decided by the action σ.
- $\lfloor\rfloor$ **:** This operator is called *timeout operator*. It is similar to the context $+, \sigma and \tau$ but is properly decided by the passage of time in favor of the right hand operand.
- $|$ **:** The parallel bar here is the handshake and interleaving bar of CCS. However, σ again behaves differently. When two process traverse time there composition also does.
- $\backslash a$ **:** This is just the restriction operator of CCS.
- $[]$ **:** This is just the relabeling operator of CCS.

The operational semantics of TPL is given in two parts. The first part given below, defines the next state relations, $\xrightarrow{\alpha}$, for each $\alpha \in Act$. This is a slight generalization of the standard operational semantics for CCS and the new action σ plays no role.

$$ACT_1 : \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$SUM_1 : \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \quad SUM_2 : \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$THEN_1 : \frac{P \left\lfloor \xrightarrow{\alpha} P'\right.}{\lfloor P \rfloor (Q) \xrightarrow{\alpha} P'}$$

$$COM_1 : \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad COM_2 : \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$COM_3 : \frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P'|Q'}$$

$$REL_1 : \frac{P \xrightarrow{\alpha} P'}{P[S] \xrightarrow{S(\alpha)} P'[S]} \quad RES_1 : \frac{P \xrightarrow{\alpha} P', \alpha \notin \left\{b, \bar{b}\right\}}{P \backslash b \xrightarrow{\alpha} P' \backslash b}$$

$$REC_1 : \frac{t[recx.t/x] \xrightarrow{\alpha} P'}{recx.t \xrightarrow{\alpha} P'}$$

The relational $\xrightarrow{\sigma}$ is defined in terms of the following relations:

$$ACT_2 : \frac{}{a.P \xrightarrow{\sigma} a.P} \quad NIL : \frac{}{nil \xrightarrow{\sigma} nil}$$

$$WAIT : \frac{}{\sigma.P \xrightarrow{\sigma} P}$$

$$SUM_3 : \frac{P \xrightarrow{\sigma} P', Q \xrightarrow{\sigma} Q'}{P + Q \xrightarrow{\sigma} P' + Q'}$$

$$THEN_2 : \frac{P \xrightarrow{\tau} \not\rightarrow}{\lfloor P \rfloor (Q) \xrightarrow{\sigma} Q}$$

$$COM_4 : \frac{P \xrightarrow{\sigma} P', Q \xrightarrow{\sigma} Q', P|Q \xrightarrow{\tau} \not\rightarrow}{P|Q \xrightarrow{\sigma} P'|Q'}$$

$$REL_2 : \frac{P \xrightarrow{\sigma} P'}{P[S] \xrightarrow{\sigma} P'[S]} \quad RES_2 : \frac{P \xrightarrow{\sigma} P'}{P \backslash a \xrightarrow{\sigma} P' \backslash a}$$

$$REC_2 : \frac{t[recx.t/x] \xrightarrow{\sigma} P'}{recx.t \xrightarrow{\sigma} P'}$$

The details explanation of TPL is available in [**?**].

4.5 ATP

ATP is also an timed processes algebra, proposed in [**?**]. This language is almost same as TPL only the syntex are different.

4.6 CSA

4.7 Timed CSP

The language of CSP [**?**] and its semantic models are appropriate for describing and analysing systems in terms of there possible sequences of events. All of these models have directly abstracted away concerns about timing such as the precise time at which events occur or the delay before an event is made available. There may be requirement to analyse a CSP process description with regard to its real-time behaviour, to understand how long executions may expected to last, or to check conditions for scheduling. To handle these, Timed CSP has been introduced [**?**]. Timed CSP is also a huge language like CSP. Hence we just dicussed the syntaxes which are very basic and usefull for our purpose.

4.7.1 Event transitions

The labelled transition $Q \overset{\mu}{'}$ will be understood at the timed level as asserting that the timed CSP process Q may immediately and instantaneously perform event μ and behave subsequently as the timed process Q'.

4.7.2 Evolution transitions

The passage of time will be described by an evolution transition $Q \overset{d}{\rightsquigarrow} Q'$, where d can be any strictly positive real-time value: $d > 0$. This is understood as Q progressing simply by allowing time to pass, so that after a delay d, it will have evolve to process Q'. The transition takes d unit of time, in that the time on the global clock will be advanced by d units of time during the course of this transition. No internal or external event is performed at any point during this transition.

4.7.3 Event prefix

The CSP expression $a \rightarrow Q$ describes a process which is prepared to engage in the event a, after which it will behave as Q. It remains in its initial state untill the event a is performed, continually offering this event to its environment untill the offer is

accepted. The two rules for event prefix are:

$$\frac{}{(a \rightarrow Q) \xrightarrow{a} Q}$$

$$\frac{}{(a \rightarrow Q) \rightsquigarrow (a \rightarrow Q)}$$

4.7.4 Successfull termination

The CSP process $SKIP$ is the immediately terminating process. Since the event $\surd$ is used to denote termination, the transitions for $SKIP$ are firstly that it is immediately ready to perform $\surd$, and secondly that it remains ready as times passes.

$$\frac{}{SKIP \xrightarrow{\surd} STOP}$$

$$\frac{}{SKIP \overset{d}{\rightsquigarrow} SKIP}$$

4.7.5 The process $WAIT$

The process $WAITd$ can perform no events for the first d units of its execution, but after delay it terminates and its subsequent behavior is that of $SKIP$.

4.7.6 External choice

The external choice operator offers a choice between processes, which is resolved at the instant the first visible event occurs, in favour of the process which performs it. In constrast to the prefix choice construct, which offers a stable choice between events, the external choice operator allows each of its arguments to execute independently, making events available and withdrawing them as time progresses, until the choice resolved. The rules for deriving transitions of a choice are given as follows:

$$\frac{Q_1 \xrightarrow{a} Q_1'}{Q_1 \Box Q_2 \xrightarrow{a} Q_1', Q_2 \Box Q_1 \xrightarrow{a} Q_1'}$$

$$\frac{Q_1 \overset{d}{\rightsquigarrow} Q_1', Q_2 \overset{d}{\rightsquigarrow} Q_2'}{Q_1 \Box Q_2 \overset{d}{\rightsquigarrow} Q_1' \Box Q_2'}$$

$$\frac{Q_1 \xrightarrow{\tau} Q_1'}{Q_1 \Box Q_2 \xrightarrow{\tau} Q_1' \Box Q_2, Q_2 \Box Q_1 \xrightarrow{\tau} Q_2 \Box Q_1'}$$

The choice $Q_1 \Box Q_2$ has as its initial offers all events offered by either Q_1 or Q_2.

4.7.7 Recursion

Recursive process definitions $N = Q$ consist of a process name N, and a CSP process description Q which may contain instances of that name. The original untimed transition rule allowed N precisely those transitions that could be deduced for Q. This is also the approach taken in the timed setting, where evolutions are allowed in addition to transitions. The rules are:

$$\frac{Q \xrightarrow{\mu} Q'}{N \xrightarrow{\mu} Q'}[N = Q]$$

$$\frac{Q \overset{d}{\rightsquigarrow} Q'}{N \overset{d}{\rightsquigarrow} Q'}[N = Q]$$

The recursive call is modelled as taking no time: the result of dereferencing the process name N has exactly the same transitions as N itself. Any delay required for the recursive calls should be explicitly included in the relevant process description.

4.7.8 Concurrency

The event transition of concurrency in timed CSP are identical to the untimed transitions. The concurrency simply requires that the time progresses at the same rate in both Q_1 and Q_2.

$$\frac{Q_1 \xrightarrow{\mu} Q_1'}{Q_1||Q_2 \xrightarrow{\mu} Q_1'||Q_2}[\mu \in (\alpha Q_1 \cup \{\tau\} \setminus \alpha Q_2)]$$

$$\frac{Q_2 \xrightarrow{\mu} Q_2'}{Q_1||Q_2 \xrightarrow{\mu} Q_1||Q_2'}[\mu \in (\alpha Q_2 \cup \{\tau\} \setminus \alpha Q_1)]$$

$$\frac{Q_1 \xrightarrow{a} Q_1', Q_2 \xrightarrow{a} Q_2'}{Q_1||Q_2 \xrightarrow{a} Q_1'||Q_2'}[a \in (\alpha Q_1 \cap \alpha Q_2)\surd]$$

$$\frac{Q_1 \overset{d}{\rightsquigarrow} Q_1', Q_2 \overset{d}{\rightsquigarrow} Q_2'}{Q_1||Q_2 \overset{d}{\rightsquigarrow} Q_1'||Q_2'}$$

Two processes can synchronize on an event they both perform only at a time when both are ready.

4.7.9 Interleaving

The concepts of interleaving is same as the concepts of interleaving in CSP. We have just given time transition rule:

$$\frac{Q_1 \overset{d}{\rightsquigarrow} Q_1', Q_2 \overset{d}{\rightsquigarrow} Q_2'}{Q_1|||Q_2 \overset{d}{\rightsquigarrow} Q_1'|||Q_2'}$$

4.7.10 Event Hiding

When an event is made internal, then the environment of the system is no longer required to participate in that event. This is same as the concelment operator of CSP.

4.7.11 Sequential composition

Here also the original concepts of CSP are same, the only new thing is the time transition as given bellow:

$$\frac{Q_1 \overset{d}{\rightsquigarrow} Q_1', \neg(Q_1 \overset{\surd}{\rightarrow})}{Q_1; Q_2 \overset{d}{\rightsquigarrow} Q_1'; Q_2}$$

The rule for evolution of $Q_1; Q_2$ has a negative premiss concerning the transitions of Q_1, as did the rule for hiding. This is used to ensure that evolutions cannot occur when termination is possible.

4.7.12 Comparision between CSP and timed CSP

Timed CSP is just an extension of CSP. Hence all the operation of CSP are valid in timed CSP. The only new thing added to all the operators is the time transition. Since both CSP and timed CSP are huge language, it is not possible to explaine every thing of the languages here. Readers can follow [**?**], [**?**], [**?**], for the complete reference. There are some concepts of CSP we use in the next chapter as timed CSP also. Though we have not given the detaile explanation about the validity of all these operations in this chapter, it is clearly explained in [**?**].

4.8 Limitations of Different Process Algebra Languages

In this section we will discuss the limitations about all the above mentioned languages. Since our aim is to construct a model for Multiple Clock Latency Insensitive System (MCLI System [**??**]), the limitations we mentioned here are according to our purpose. There may be some other limitations also:

- **CCS:** The limitation of CCS is that it has not mentioned any thing about the passage of time in a system. Here it is not possible to model a timed system with CCS.
- **TPL and ATP:** Both TPL and ATP are the extension of CCS and developed to model timed systems. They allows time transitions. But they are only applicable in single clock system. A distributed clock system (e.g, MCLI system) cannot be modeled, with these two languages.
- **CSA :** This language is extended from TPL to model multiple clock systems. This language can model multiple clocks but it has the following limitations:
  - Sequential operator is not mentioned; CSA has not mentioned any thing about successful termination of a process. Hence, there is no sequential processes in CSA.
  - Conditional operator (if-then-else) is not mentioned.

  Though we cannot say that it is not possible to model our MCLI system in CSA, it is very complex to model it without the above two concepts. Our main aim is to extend the model of single clock LI systems, proposed in [1]. The author of [1] modeled it in CSP using both the above concepts. Even these two concept are very necessary for modeling MCLI systems.

  It may be possible to model the MCLI system in CSA but we have not chosen it because of facing problem due to the absence of *sequential* and *if-then-else* operators.

Timed CSP: Timed CSP has also not mentioned any thing about multiple clocks. But the time passes with reference to a single conceptual global clock. But it can be used to model multiple clock system as explained in the next section.

4.9 Why we have chosen Timed CSP

The main reason of choosing timed CSP is that it is a complete language and offers a large amount of operations to full-fill our need. On the other hand, timed CSP passes time with reference to a single conceptual global clock. Time progresses at the same rate in all components of the system, as measured by this global clock. The time is modeled as the non-negative real numbers. In MCLI system there may be multiple clock and each clock has different clock period. But we can measure their clock periods with the global-clock of timed CSP. For example, consider that there are two clocks C_1 and C_2 in an MCLI system. Now we can measure the clock period of both the clocks with the global-clock. It means that the clock period of C_1 may be x time unit of the global-clock and the clock period of C_2 may be y time unit of the global-clock. So it is possible to model different clocks in timed CSP with different clock periods.

4.10 CSP Model of Single Clock LI System [1]

In this section we have explained the CSP model of single clock LI system, proposed in [1]. The MCLI systems also use the modules used in single clock LI systems (cf. section **??**), hence we have also reused the model of different modules of single clock LI system, in our model of MCLI system. The detail explanation about the modeling of the single clock LI system can be found in [1]. We have just given a brief explanation about the model.

As mentioned in [2], single clock LI system consists of two types of modules, (a) IP core with an encapsulation circuit (LI Computational Block) and (b) Relay Stations (LI Connectors). In [1], the modifies some basic concepts proposed in [2] and gives the CSP model for both LI Computational Blocks and LI Connectors. For modeling them, the author use CSP, because the single clock LI system is a completely synchronous system. She therefore use the discrete time version of Timed CSP that has the tock signal which is a timed event occurring at fixed time intervals. Instead of adding time to the semantic model, she used fixed-interval time event as just another event ($tock$) in the untimed version of CSP. The use of this tock event is to denote passage of one unit of time. To specify synchronous systems, there is an underlying assumption that all events between two successive $tock$ events can be completed within the given

time interval.

4.10.1 Modeling LI Computational Block

The IP core we used for making the single clock LI systems may not be in a position to handle arbitrary delays in its input-output channels [2]. Hence to make them patient it is required to encapsulate them with a wrapper circuit. This procedure is called encapsulation [2]. This encapsulated circuit is called as LI Computational Block. It is described by a process with input and output ports (Figure **??**). This process being synchronous waits for the clock tick ($tock$ event) before accepting inputs and producing outputs.

However, the process may/may not be in a position to produce output at each tock. The main reasons for this are

1. all the necessary inputs are not available;
2. there is not enough room to store outputs on the downside channels;

The input and output ports are divided into some channels, as mentioned below,

1. Channel carrying the data to the process: $a.dIn$ or $b.dOut$.
2. Channel carrying the information about data (in)validity: $a.void$ or $b.void$..
3. Channel carrying the information that data are not consumed: $a.stall$ or $b.stall$.

The direction of the data carrying channel is decided depending upon whether the port is an input port or an output port.

Three channels of the input port: If the environment has not put valid data on the channel, then it must inform this to the process over the void channel. If the process receives a "1" on this channel (i.e., $a : void?1$), then the process understands that data are not valid. This is an indication by the sender for the process to stop computing new results; however, interaction over the channels can still continue. A value of "0" on this channel (i.e., $a : void?0$) indicates that the data are valid and ready for use. The author however model this by synchronization either on the data channel or the void channel. Which of the two signals (data/void) is received depends on the sender. Thus, an event on void channel means that data are absent, while an event on the data channel reads the datum. These values of "0"

or "1" are helpful in the implementation of such a model where every channel (wire in the implementation) must have some signal being transmitted. However, one can abstract this information to simple synchronizations either on the data channel or on the void channel. Therefore, we simply have an event on the void channel (instead of sending a "0" or "1") representing a void event. If data are not available on all inputs, the process stops computing new results until they are made available. Due to this, there could be certain input ports with valid (unconsumed) data. In this scenario, we need to prevent the senders of these channels from sending newer datum to avoid data loss due to overwriting. This feature is enabled by sending a "1" over the third channel of the input port: $a : stall!1$.

Three channels of the output port : Output not being generated at each $tock$, the process needs to inform its receivers about the invalidity (unavailability) of data on its output data channels. This is done by sending an event on the void channel of the output port. An event $b : void$ indicates to the receiver that data on data channel of b are not valid (in this case, actually, no event is sent on the $dOut$ channel). Similar to the inability of process P to consume data on its input port, the receiver of the data sent by P may not be able to consume it. The receiver can therefore send a stall to $P : b : stall?1$.

Thus, an LI module is able to function (read input and generate output) under a synchronous setting, provided the environment behaves in a certain way (i.e., environment sends appropriate void and stall signals). However, to ensure fairness, the environment is assumed not to send void and stall signals indefinitely.

In a synchronous module at the nth $tock$, the process emits the outputs produced during the $(n-1)$th $tock$ and consumes inputs to compute the output to be emitted at the $(n+1)$th $tock$. It thus has a state to remember. To model this, the author uses process, which has state. This state can be map to an internal storage element at the implementation stage. The process behavior uses a set of state variables (i.e.,state up to the previous clock cycle) which are explained below:

- x**:** Array to hold state of the data inputs received. It is overwritten whenever new data are read. Size of array is equal to the number of input ports and these ports are identified by their indexes.

- xa**:** Auxiliary array for x. In the case when data from x are not consumed, new data are stored in xa. At the end of computation cycle, this array is used

to update x.

- y: Array storing the outputs computed during the previous clock cycle. Size of array is equal to the number of output ports and these ports are identified by their indexes.
- s: Array denoting the inputs received on the stall channels associated with the output ports. The array size of this is the same as that of y.
- xnw: Array storing information if the values on the corresponding input ports are a new value, i.e., a value which was not used to compute a result. At most, two new values can be present on each input port. xnw is updated after the output values are computed.
- st: Boolean variable (1 =true, 0 =false). This variable indicates if a stall request was received during the previous clock cycle.
- ynw: Boolean variable. This tells if the value in the data array y is newly computed ($ynw = 1$) or is a stale value ($ynw = 0$).

Now the LI Computational Block is denoted by a process defined as follows:

$$
\begin{aligned}
P_{(x,xa,y,s,xnw,st,ynw)} &= tock \rightarrow \\
&\quad ((STLOUT|||IN) \rightarrow \\
&\quad\quad (R11 \lhd \exists j.s[j] = 1 \rhd R01) \\
&\quad) \\
&\quad \lhd st = 1 \rhd \\
&\quad ((DTOUT|||IN) \rightarrow \\
&\quad\quad (NXTSTL \lhd \exists j.s[j] = 1 \rhd REPEAT) \\
&\quad)
\end{aligned}
$$

$$
STLOUT = (|||_{j \in \mathcal{B}} j.void!) \rightarrow (|||_{i \in \mathcal{A} \wedge xnw[i] \neq 0} i.stall!1)
$$

$$
\begin{aligned}
DTOUT &= (|||_{j \in \mathcal{B}} j.dOut!y[j]) \\
&\quad \lhd ynw = 1 \rhd \\
&\quad ((|||_{j \in \mathcal{B}} j.void!) \rightarrow (|||_{i \in \mathcal{A} \wedge xnw[i] \neq 0} i.stall!1))
\end{aligned}
$$

$$
\begin{aligned}
IN &= (|||_{i \in \mathcal{A}} ([i.void? \\
&\qquad \Box \\
&\qquad (i.dIn?x[i] \lhd xnw[i] = 0 \rhd i.dIn?xa[i]) \rightarrow \\
&\qquad\quad xnw[i] = xnw[i] + 1 \\
&\qquad]) \\
&\quad) \rightarrow (|||_{j \in \mathcal{B}} [j.stall?s[j] \Box SKIP])
\end{aligned}
$$

$$
\begin{aligned}
NEXTSTL &= P11 \lhd \forall i.xnw[i] \neq 0 \rhd P10 \\
REPEAT &= P01 \lhd \forall i.xnw[i] \neq 0 \rhd P00 \\
R01 &= P_{(x,xa,y,s,xnw,0,ynw)} \\
R11 &= P_{(x,xa,y,s,xnw,1,ynw)} \\
P00 &= P_{(x,xa,y,s,xnw,0,0)} \\
P01 &= P_{(x',xa,y',s,xnw',0,1)} \\
P10 &= P_{(x,xa,y,s,xnw,1,0)} \\
P00 &= P_{(x',xa,y',s,xnw',1,1)}
\end{aligned}
$$

where $y' = f(x)$ is the result of the computation performed by the block using inputs obtained in the current clock cycle. y' is an array where each entry corresponds

to the output to be made on the corresponding output channel. The array x' is updated as follows:

if(xnw[i]=2)
$then\,\{x'[i] = xa[i]; xnw'[i] = 1;\}$
$else\,\{x'[i] = x[i]; xnw'[i] = 0;\}$

Whether the contents of xa are fresh/stale is decided by the variable xnw. Values in xa are copied to x for the new iteration, and hence, the array xa need not be updated (it can simply be overwritten).

The complete explanation about this model is given in [1].

4.10.2 Modeling LI Connectors

If the signal takes more than one clock cycle to reach the destination, the receiver will malfunction, as it will not get the correct inputs signals at the correct time. This is because the receiver needs some input at each clock tick, the input may be a data value or a void signal. It is therefore necessary to modify these connecting wires which are capable of delivering either correct data or void signals to the receiver at every clock tick. The connecting wires therefore need to be replaced by buffers. These are called relay stations in [2] that can be abstracted as distributed FIFOs.

The main idea being a storage element with some intelligence. The function of a storage element is to read input, store it, and make it available at the output. However, to have a storage element which is LI, we need to modify its behavior. The element must also take care of stall and void signals in cases where the receiver is unable to consume the stored data. This is described by the following definition. The definition is similar to that of the computational block. The difference being on the input-output channels. The storage element has only one input and one output port. Thus, we do not have x, xa, y, s as arrays but as single variables. We still need the variable ynw to denote new value on the output. Let i denote the input port and j denote the output port.

where,
if(xnw==2)
$then\,\{x' = xa; xnw' = 1;\}$
$else\,\{x' = x; xnw' = 0;\}$

This being a connector, no computation takes place and so the input value is forwarded as the output value. If input data are to be output, it is copied ($C01$ and

$C11$), otherwise the old value is retained ($C00$, $C10$, $R01$, and $R11$).

4.10.3 Analysis and Properties

The author [1] proves that her model satisfies all the properties of single clock LI systems. She also proves that there is no chance of data loss or deadlock in the proposed models. We use these the model of these two modules (LI Computational Block and LI connector) in our modeling of MCLI system (see chapter **??**).

Chapter 5

Proposed Model

5.1 Introduction

In this Chapter we propose a process algebric model for Multiple Clock LI Systems. We use the concepts of Timed CSP [][see **??**] to model the system.

5.2 The block diagram of the Multiple Clock Relay Station (MRS)

In the theory of multiple clock LI design (see Chapter 3), we have given a theoretical definition of MRS (see definition 3.5.4 and 3.5.5). We have also given the properties of the MRS and proved that it is a multiple clock patient process. Hence, MRS is a Multiple Clock Latency Insensitive (MCLI) process. In this section we have given a block diagram for the MRS, explained in Chapter 3. We have designed this block diagram based on the mixed-clock FIFO design proposed in [**?**]. The MRS is divided into three main parts,

- **Input Interface:** This part has the same clock as the sender of the MRS. Here "sender to the MRS" means the module which sends data to the MRS. Henceforth, we call it as **Sender**. The purpose of this part is to take the input from the **Sender** and store them into the storage of the MRS. If in any clock cycle, the **Sender** has not send any valid data then it will send an invalid signal through the channel $i.void$. The channel $i.dIn$ is used for data input.

In addition to this, the interface will send a stall request $(0/1)$ to its **Sender**. The request will send 1 if the storage is *full* and 0 otherwise.

- **Output Interface:** This part has the same clock as the receiver of the MRS. Here "receiver of the MRS" means the module which receives data from the MRS. Henceforth, we call it as **Receiver**. The purpose of this part is to send the information from the storage of the MRS to the **Receiver**. However if it (MRS) has received any stall request from the **Receiver**, in its previous cycle or the storage of the MRS is empty then it will send invalid data by asserting $j.void$ channel. Hence at every clock cycle the output interface will send valid/invalid data and also take stall request $(0/1)$ as input, from the **Receiver**.

- **Storage:** The storage of the MRS is an array of cells. Hence, capacity of the MRS means the number of cells in the MRS.

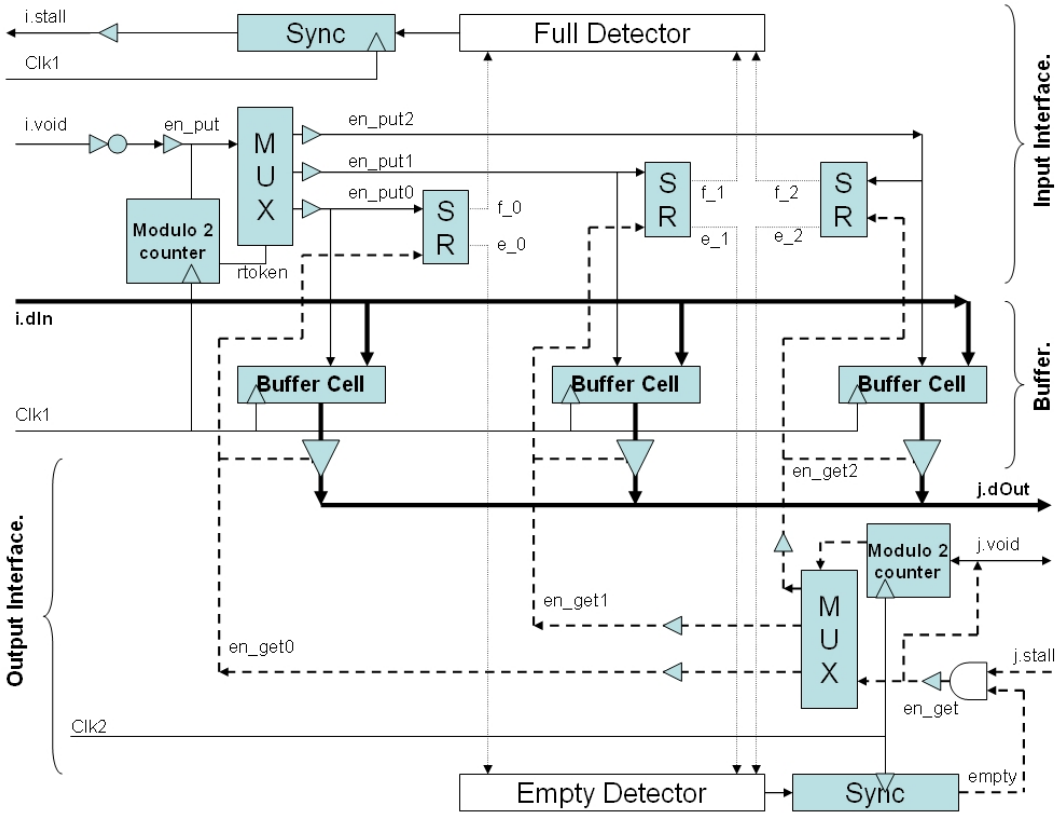

Figure 5.1: *The Block Diagram for MRS.*

The complete block diagram of the MRS having capacity 3 is given in the figure 5.1. Now we explain the entire block diagram step-by-step.

5.2.1 Input Interface

As we already mentioned that the purpose of this interface is to receive valid/invalid data from the **Sender** and store them into a particular cell provided the data is valid. Also it sends a 1, as a stall request to its **Sender** if the storage of the MRS is full, otherwise sends a 0. Now to do this task in every clock cycle, the input interface is itself divided into some blocks. Note that in MRS the input interface and the output interface are running in two different clocks. Hence, in this section the clock cycles we mention are actually the clock cycles of the clock belonging to the input interface.

- **Cells:** This are the storage elements of the MRS. Input interface will store the valid input data into the cells. A cell can only store data when its enable line is high. The interface will make only one cell enable at a time.
- **Data bus:** When the sender sends a data, it will become available in the data bus ($i.dIn$). Note that the data bus is common for all the cells in the in the MRS. Which cell will store this data, is depend on a token, called *receiver-token*.
- **Modulo 2 Counter:** This is a counter starting initially from zero and again back to zero when it increment after 2. The purpose of this counter is to generate the *receiver-token*. In every cycle, the value of the counter will decide, which cell of the MRS will store the data available in the data bus at the starting of the next cycle. The counter value will only change after the cell indicating by the current counter value, has been filled by a data. For this reason, an enable signal is connected to the counter, which means that the counter can only update its value at those clock cycles where the enable line is high.
- **Full Detector:** The full detector is an asynchronous block. Its output 1 if the MRS is full, otherwise 0. The status of the MRS storage can be changed in the following two ways:
  - **When a data stores into a cell:** the data stores into a cell during the rising edge of a clock cycle. Hence we can say that this change is synchronous with the input-interface clock.

  - **When a data removes from a cell:** the data remove from the MRS by the output-interface, which is running on different clock. Hence, this change is asynchronous with respect to the input-interface.

  To handle the status-change $(full/empty)$ of the MRS storage, the *full detector* is actually an asynchronous block. Since *full detector* is an asynchronous block, its input cannot be used directly by input-interface. A synchronizer is needed to synchronize the output of the *full detector*.

- **SR Latches:** The SR latches in the diagram are asynchronous flip-flops. Each cell of the MRS has a separate SR latch. When the cell is full, the latch will be in a state of $f_0 = 1$ and $e_0 = 0$. As soon as the cell become empty, the latch will change its state to $f_0 = 0$ and $e_0 = 1$. There is no chance in our design that both the *set* and *reset* signals of an SR latch will become 1 simultaneously.

- **Synchronizer:** As we already mentioned that the *full detector* is an asynchronous block, hence we have to synchronize its output before sending it to the **Sender**. The synchronizer we use is a 2*Flip-Flop* synchronizer. Its purpose is to take the full detectors output as input and send it to the **Sender**. This signal will actually works as a stall request for the **Sender**. If the request is 1 then it means that the MRS is full and the **Sender** must not send any valid data at the next clock cycle.

- **Multiplexer:** The purpose of this multiplexer is to enable the appropriate cell according to the current counter value (*receiver-token*).

Now we will explain the detail operations of the *input-interface*. As we already mentioned that the **Sender** will send a 1 through the channel $i.void$ if the data is not valid in that cycle. But when the **Sender** sends a valid data through the data bus, its also sends a 0 through the channel $i.void$. Its means that the data in the data bus is not invalid.

Consider that the **Sender** has send a valid data at the positive edge of the nth clock tick. Since the **Sender** is sending a valid data, it will send a 0 through the channel $i.void$. Now the data becomes available in the data bus named $i.dIn$, before the starting of the next cycle and the data will be stored in a cell by the positive edge of the next cycle. But in the n^{th} cycle, the *input-interface* has to enable a particular

cell of the MRS, in which the data will be stored during the starting of the next cycle ($n + 1^{th}$ cycle). Only one cell should be enabled. As we already mentioned that we are assuming the cells in the MRS as an array of cells hence, each cell has a array index. Now only that cell will be allowed to enable, whose index is matching with the *receiver-token*. Means, the cell which currently holding the *receiver-token*, will be enabled for storing the data (available in the data bus at the n^th cycle) at the positive edge of the $n + 1^{nt}$ cycle. The selection procedure is done by the MUX. The input of the MUX is the negate of $i.void$. Hence at the n^{th} cycle it will be 1. Now if we assume that the current value of the counter is 0, then MUX will pass the input 1 through the channel en_put0, which will then enable the 0th cell of the MRS. The remaining output channels of the MUX will pass a 0, hence the other cells cannot be enabled.

Now enabling a cell means it will store the data available in the data bus at the positive edge of the next cycle. But since the cell is becoming full, the information is also necessary to send to the *full-detector*. Hence, the value of the en_put0 is also send to the corresponding SR latch of the 0th cell. Since the input is 1, the SR latch will output a 1 through f_0 and 0 through e_0. Note that all the operations up to inserting 1 to the SR latch is done during the nth clock cycle. There is one more task remaining for the nth cycle. It needs to enable the counter. Hence at the positive edge of the next cycle the counter value can also change. As we know, both SR latch and the FULL detector are asynchronous blocks, hence the state of SR latch and *full-detector* will change asynchronously. Though we can say that, after inserting the value of en_put0 at SR latch, the *full-detector* will change its state at the end of the nth clock cycle only. We can assume that because both SR latch and the full detector are build with asynchronous flip-flops. The output of the *full-detector* is taken by the **synchronizer** block. Synchronizer will take the status of the *full-detector* at the positive edge of each cycle. It will take the very current status of the *full-detector*. Remember, the status of the full detector is not singly depends on data input but it also depends on the data send by the *output-interface*. Hence, the status of the *full-detector* will change asynchronously and the **synchronizer** will take the current status, during each positive edge of its clock.

The **synchronizer** will send the $full/empty$ information at the positive edge of every clock cycle. If at any clock cycle, the synchronizer sends a 1 to the **Sender** then the **Sender** cannot send any valid data at the next clock cycle. Because, synchronizer

sends 1, means all cells of the MRS are full and there is no place for storing another data. Hence, the **Sender** sends data, depending on the stall request received from the **synchronizer**, during the previous cycle.

If the **Sender** sends a invalid data during the positive edge of the nth clock cycle, then it will also sends a 1 through the channel $i.void$. In this case none of the cells will be enable and also the counter will not enable. Hence nothing will store in the MRS during the positive edge of the next cycle. Also the counter will not increment.

There is some synchronization issues which we will discuss after the description of *output-interface*.

5.2.2 Output-Interface

The *output-interface* sends data to the **Receiver**. It receives the stall request from its **Receiver**. It only allows to pass the valid data to the **Receiver** if and only if the current cycle has not received stall request as 1 and the MRS is not empty. Otherwise it send 1 through $j.void$, means that the data in the output data bus is not valid. As shown in the figure 5.1, the *output-interface* is belongs to $clck_2$, which is different from the corresponding one of *input-interface*. The *output-interface* is also divided into some blocks as mentioned below.

- **Output Bus:** The *output-interface* sends data through this bus, and this output bus become the input bus for the **Receiver**. Hence if data become available in the output bus ($j.dOut$) at the end of the current clock cycle, then the **Receiver** can receive the data at the positive edge of the next clock cycle. Only one cell of the MRS can be able to pass its data through the *output bus*, at a time.

- **Tristate buffers:** The purpose of these *tri-state buffers* is to allow the content of the cells to pass through the *output bus*. As soon as the data has been stored into a cell, it became available at the output line of the cell. The *tri-state buffer* will prevent it to pass through the *output bus*. Only those cell can pass its data which index is matching with the *sender token*.

- **Modulo-2 counter:** The is a counter same as the counter mentioned in *input-interface*. The only difference is that it operates on the clock $clck_2$ and its input is depend on both the stall request and the empty status of the MRS.

In every cycle the value of the counter will decide which *tri-state buffer* should be enabled for passing the data through the *output bus*. Only one buffer will enable at a time and this is done by the value of the counter. The value of the counter will only change if a valid data is send by the MRS.

- **Empty detector:** An asynchronous block doing just the opposite work of the full detector used in the *input-interface*.
- **Synchronizer:** This *synchronizer* is running on the clock frequency of $clck_2$ and its outputs the synchronized empty signal (1 for empty and 0 for non-empty).
- **MUX :** The purpose of this MUX is to choose the *tri-state buffer*, which must be enable at the current clock cycle.
- **SR Latch:** The same set of SR Lateches mentioned in 5.2.1, also used here.

Now we will explain the detail operations of *output-interface*. Note that the interface will allow to pass a valid data through the *output bus*, if and only if the interface has not received any stall request (with value 1) or an empty input (with value 1), at the positive edge of the current cycle. Otherwise, it will send a 1 through $j.void$. In this case no valid data will newly placed in the *output bus*.

Consider the counter value is 0. Its means that the *sender token* is 0 and the data of the 0th cell will be send next. Now also consider that, at the positive edge of the nth cycle (clock $clck_2$) the interface receives a 0 through the stall request channel and also receives a 0 from the *synchronizer*. This means that the **Receiver** is ready to take new data and also the MRS has data, to send. Hence en_get will become 1 when both $jStal = 0$ and $empty = 0$. In this case (at the nth cycle), it is 1. Now the value of en_get will pass through the MUX according to the counter value. Since the counter value is 0, hence the value of en_get will pass through the en_get_0 and finally enables the *tri-state buffer* of the 0th cell. Hence the data of the 0th cell will become available during the end of nth clock cycle. Hence, at the positive edge of the next clock cycle, the **Receiver** can receive the data from its input bus. Sending a data from a cell means the cell will become empty, hence it may change the status of both empty and full detectors. Hence during the nth clock cycle, when the value 1 will become available at en_get_0, it will also inputed into the corresponding SR latch

of the 0th cell. The SR latch will then change its state as $f_i = 0$ and $e_i = 1$. The status of the remaining latch will remain same. As a result of this the status of both the *full detector* and *empty detector* may change.

5.2.3 Synchronization issues:

In this section we will discuss some synchronization issues of our proposed MRS design. We need to do some modifications in the above design. The techniques are mentioned in [**?**]. We are just using their proposed techniques [**?**] in our design.

5.2.3.1 Modification of full and empty detector

The added latency through the synchronization may cause the MRS to overflow or underflow. For example, when the MRS become full, the sender module will receive the information two clock cycle later; so in the next clock cycle the **Sender** might send a valid data. A similar problem arises when the MRS becomes empty.

The solution to the over/underflow problem is to change the definition of full and empty dectectors. The MRS is now considered "full" when there are either 0 or 1 cells filled. Thus, when there are fewer than 2 data items, the MRS is declared empty. The new definitions do not change the protocol with the systems. The only effect is that sometimes the system see an $n-$place MRS as a $n-1$ place one.

5.2.3.2 Deadlock avoidance:

Unfortunately, the early detection of empty, in some case, may cause the MRS to deadlock. Using the new empty definition (0 or 1 data items), it is possible that the MRS still contains one data item, but the requesting **Receiver** is still stalled.

The final safe solution is, therefore, to use a bi-modal empty detector. The detector, in addition to computing the "new empty" definition (ne), also computes the "original empty" one (oe). The block diagram of both empty detectors with there corresponding synchronizers is shown in the figure 5.2. The two empty signals are then synchronized with the receiver clock and combined through an AND gate (as shown in figure 5.3) to form the global $empty$ signal.

The intuition behind the bi-model detector is as follows. If there have been any rising stall request (i.e., 1)- for at least one clock cycle-oe dominates. This is

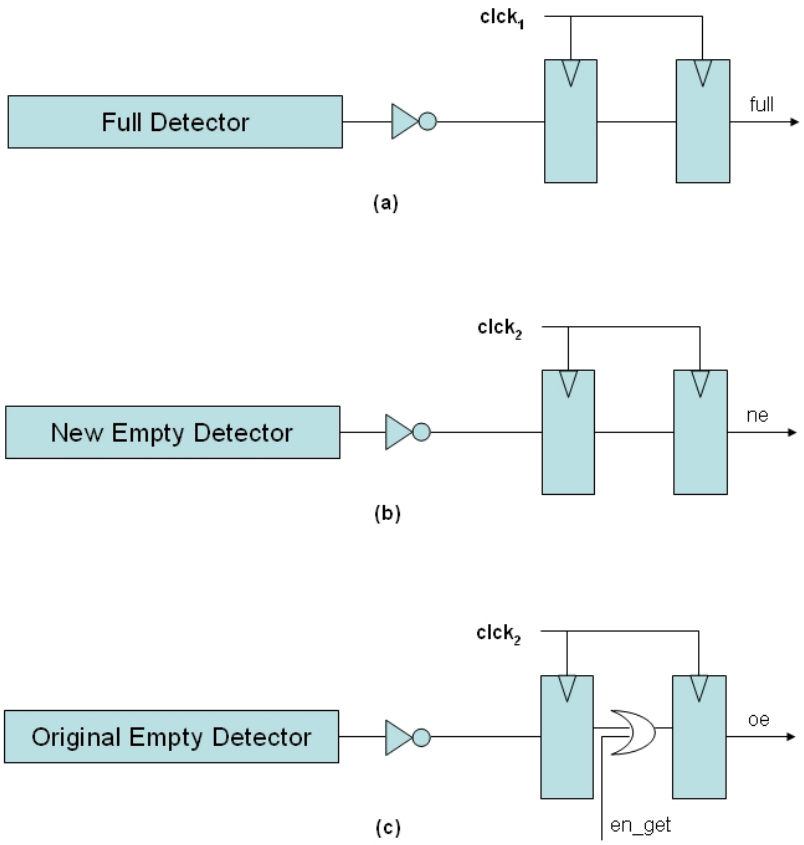

Figure 5.2: *Block Diagram of (a) Full Detector, (b) New Empty Detector and (c) Original Empty Detector.*

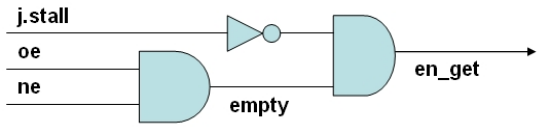

Figure 5.3: *The combinational circuit, showing the generation of final $empty$ signal.*

important when there is only one data item in the MRS: the output interface needs to send it, so oe is used to indicate the MRSs state (not empty). However, when the *output-interface* has just removed a data item, ne must be used to indicate the state, in order to prevent the MRS underflow, which the synchronization delays for oe might cause.

The "original empty" detector [Figure 5.2(c)] and the controller [Figure 5.3] im-

plements exactly the above behavior. The OR gate in the oe synchronizer is very important: controlled by en_get, it sets the oe to a neutral state ("MRS empty") one clock cycle after a valid data has been send. In this case, the "new empty" definition can take precedence in the get controller.

5.3 Some Concepts About the CSP Representation of Clocked Processes

Clocked process means the process which synchronize with some clock ticks. Process can be *single clock* or *multiple clock*. However we will show that the *single clock* processes are also *multiple clock* process (having only one clock). In this section we explain some concepts we use to model the clocked processes in timed CSP []. This concepts are very important because based on this concepts we design the model of MRS in timed CSP. Later in this section we also describe the important definitions of the theory of MCLI design (cf. Chapter 3), in terms of timed CSP.

5.3.1 Clocks :

A clock is a process with the only one alphabet $tock$. In timed CSP this clock can be defined as

$$Clock = tock \overset{d}{\nRightarrow} Clock.$$

The event $tock$ means the starting of a new cycle. After happening the $tock$ event, the process will wait for d amount of time and then again engage with a new $tock$. The process will run continuously in a recursive fashion. Now if we consider that the time d is the time period of a clock C then the above process exactly define the clock C.

In our case, we need multiple clocks. Hence we consider an array of clocks and we assume that all the clocks we need, are belongs to this array. We call this array as $Clocks$. The CSP model of each clock in the $Clocks$ will be,

$$Clocks[i] = tock_i \overset{d_i}{\nRightarrow} Clocks[i].$$

Where $i \in [1, N]$ is the index of the $Clocks$ array. $tock_i$ means the $tock$ event of the $Clocks[i]$ and d_i means the time period of $Clocks[i]$.

5.3.2 Single Clock Process:

Single clock process means the processes which can synchronize with a clock in the every rising edge of the clock cycles. The CSP model of a process P which can synchronize with the $Clocks[i]$ is defined as,

$$P = tock_i \rightarrow \ldots \rightarrow P.$$

Note that the above definition of P not correctly says that it synchronizes with $Clocks[i]$. Because, here we have not shown any concurrency between P and $Clocks[i]$. So whenever we need to define a process P synchronizing with $Clocks[i]$, we have to write $P||Clocks$. This means that the process P is running concurrently with all the clocks in $Clocks$. But since P is agreeing only with the $tock_i$ event, it will only synchronize with the $Clocks[i]$. The $tock$ events of the remaining clocks will be ignored by P.

5.3.3 Multiple Clock Process:

Multiple Clock process means the process which can synchronize with the multiple clocks of $Clocks$. The technique of defining a multiple clock process is to divide the process into some single clock subprocesses. For example if a multiple process is synchronizing with three clocks, then we have to divide the process into three single clock subprocess. The process have different set of actions under the scope of its own clock. Each single clock subprocess will take care of one action set, which is under the scope of its own clock. The only assumption we have taken here is that each subprocess is running concurrently without interacting with each other. Means the set of actions are not depended on each other. Due to this assumption the process may not be able to meet the properties of all real life multiple clock systems, but it is sufficient for our purpose.

If a process P can synchronize with three clocks namely $Clocks[i], Clocks[j]$ and $Clocks[k]$, then the process P can be defined in CSP as

$$P = P1||P2||P3$$

where

$$P1 = tock_i \rightarrow \ldots \rightarrow P1,$$

$$P2 = tock_j \rightarrow \ldots \rightarrow P2,$$

and

$$P3 = tock_k \rightarrow \ldots \rightarrow P3,$$

Note that as mentioned in section 5.3.2, whenever we need to define a process P synchronizing with some clocks of $Clocks$, we have to write $P||Clocks$.

Theorem 5.3.1. *Since multiple clock process is just an interleaved composition of single clock processes, hence we can say that a single clock process is a spacial case of multiple clock process. But having only one clock.*

5.3.4 Event Interleaving

CSP has not given any syntax for event interleaving, hence it is not possible for us to apply the interleaving operator ($|||$) of CSP for event interleaving. But we use it in a different sense as explained here. Consider that a and b are the events of a process P. Now if we write $a|||b$ then it is not valid in CSP, because according to CSP, only processes can interleaved together. But if we re-write the above expression as $a \rightarrow SKIP|||b \rightarrow SKIP$, then the syntax is completely valid because in this case both $a \rightarrow SKIP$ and $b \rightarrow SKIP$ are processes and not just events. In this case the interleaved processes will execute independently on each other and do not interact on any events apart from termination. Henceforth, if we use the interleaving operator between events then its means that we have just written it in a shorter form, ignoring the $SKIP$ process intentionally.

5.4 Modeling Multiple Clock Latency Insensitive Process

A multiple clock latency insensitive system is a collection of LI modules (processes) having different clocks. If any two LI modules having same clock, want to communicate, then this communication happens directly or through the single clock relay stations [2]. When all the processes in a subsystem are belongs to same clock without any phase difference, then the subsystem is called a Single Clock Latency Insensitive System (SCLI), proposed in [2]. A CSP modeling for the SCLI systems is already proposed in [1]. In this section we extend the CSP concept proposed in [1] for Multiple

Clock Latency Insensitive Systems (MCLI). Before going to any further details here we are explaining the basic concepts of MCLI systems. Since we already explained every thing regarding MCLI system in Chapter 3, it is just written here for the completeness purpose of this chapter.

5.4.1 MCLI systems

MCLI system is a system having a set of modules, which works correctly in spite of arbitrary delays on the connecting channels. The modules may belongs to different clocks. However each module can only belong to a single clock. Hence the MCLI system is nothing but a Globally Asynchronous and Locally Synchronous (GALS) systems. Now we explain the main steps of designing an MCLI system.

- Collect some single clock modules, not necessarily belongs to same clock.
- Make them patient by using some wrapper circuit. The detail is already explained in [2] and also in 1.1.2.
- Design the system without considering any communication delays.
- Insert Single clock relay stations (RS) in the communication channel of two modules, where modules having same clock, the communication takes more than one clock cycle delay.
- Insert a multiple clock relay station (MRS) in the communication channel of two modules, where modules having different clocks.

From the above explanation it is clear that we are using only an additional MRS with the existing SCLI modules to design MCLI systems. Hence in case of the CSP modeling of MCLI systems, we no need to change the model for *LI Computational Blocks* and *LI Connectors* as proposed in [1]. We just need to model the MRS. The resulting model of the complete MCLI system will be a model composing with *LI Computational Blocks*,*LI Connectors* and *MRS*.

5.4.2 Understanding the CSP Model of MRS

We know that the processes of CSP can be imagine as a set of black boxes connecting with each other through some channels. For example a process $(P||Q)$ having $\alpha(P) \cup$

$\alpha(P) = a, b$, can be explained by the figure 5.4. Before going to explain the details of the MRS modeling, first we are explaining it by using a block-box diagram of the model (see Figure 5.5).

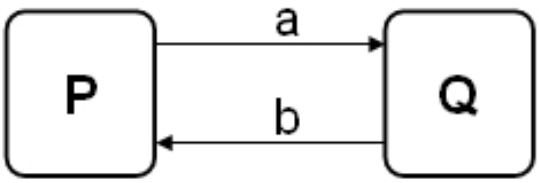

Figure 5.4: *A black box diagram showing* $P||Q$.

The detail implementation of the MRS is already explained in section 5.2. The modeling we done for the MRS is an abstract form of the block diagram given in the Figure 5.1. We assume some concepts in upper label without considering every thing shown in the Figure 5.1. The main assumptions we have taken are,

- We have not modeled each cell separately. Rather we assume the cells as a global array x, which can be access by both the input and the output interface of the MRS. The only restriction is that both input and output interface cannot access same item from x simultaneously.
- Instead of considering the Full and Empty detectors as asynchronous combinational blocks we assume them as asynchronous sequential blocks. All the detectors knows the capacity of the MRS (i.e., the maximum size of x), and the number of currently available data.

Figure 5.5 gives the complete structure of of our model, we now explain each and every process in this figure briefly.

P1: This is a single clock process (see 5.3.2). It has four channels namely $i.dIn$,$i.void$,$tockR$ and $fill$. The operation of $P1$ is explained below.

1. Since $P1$ is a single clock process, hence it can synchronize with a clock process. Hence according to the requirement, $P1$ is synchronizing with the clock $ClockR$. They are synchronizing by the common tock event $tockR$.
2. The two channels $i.dIn$ and $i.void$ are connecting the MRS with the **Sender**. At any clock cycle, $P1$ receives a valid data means it engages in $i.dIn$, simul-

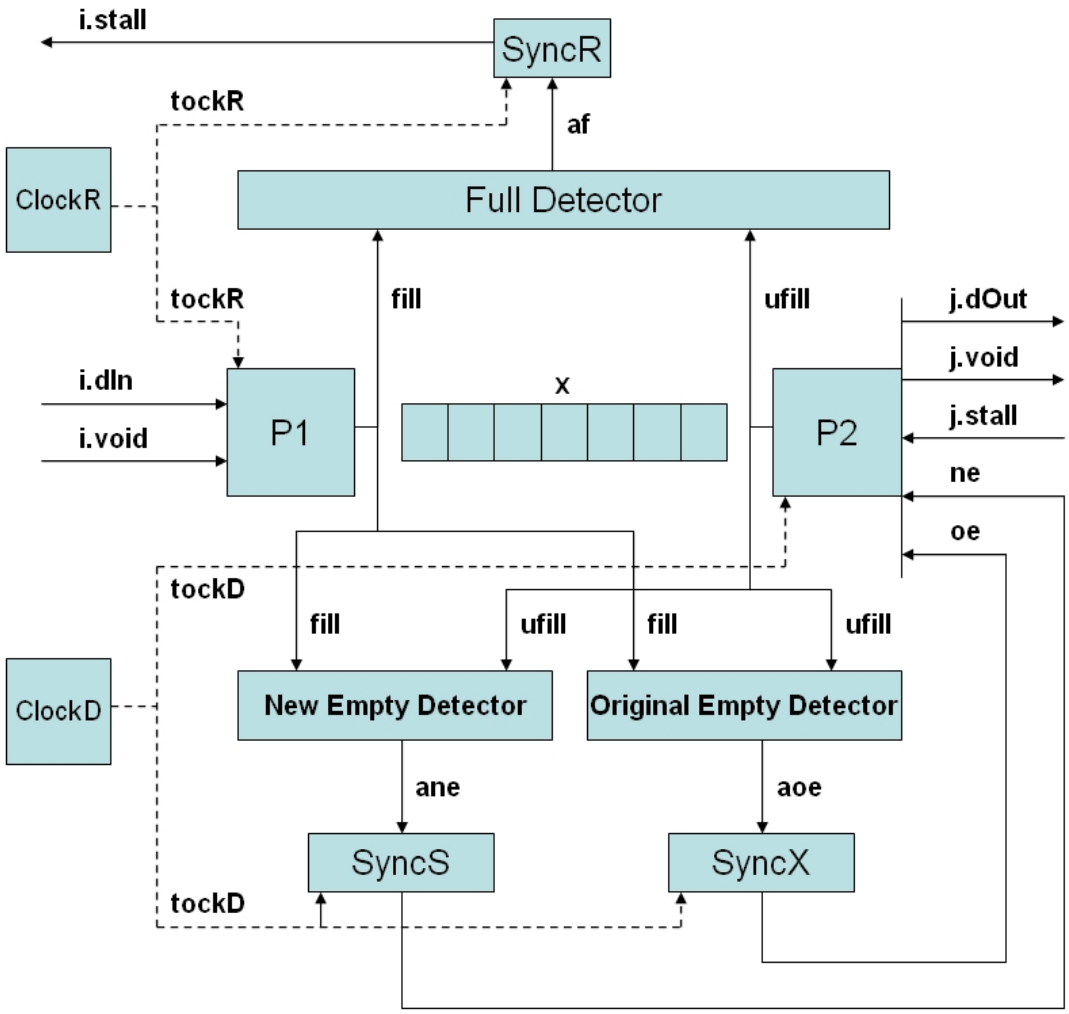

Figure 5.5: *The black box diagram of MRS in CSP.*

taneously with the **Sender**. On the other hand, at any clock cycle, $P1$ receives an invalid data means it engages in $i.void$, simultaneously with the **Sender**.

3. If the data received by $P1$ is valid then it has to store in the global array x. The position of the array where the new data should be stored is defined by a number called **receiver token**.

4. The *receiver token* will be incremented after the valid data stored in the $x[receivertoken]$. The incrementation is done such that the *receiver token* will return to 0, when it will increment from n, where n is the size of the array.

5. After storing a valid data into the array x, the status of the array (empty/full) may be change. Hence after each insertion of valid data into x, the $P1$ will send a $fill$ message to both the full and empty detectors. The full/empty detectors will also simultaneously agree with this $fill$ event.

All this work must be done within one clock cycle of $ClockR$ and after this, $P1$ will again synchronize with $ClockR$ and starts the execution of the next cycle.

P2: This is also a single clock process. It has five channels, namely $j.dOut$,$j.void$,$j.stall$,ne, oe and $tockD$. The operation of $P2$ is explained below.

1. Since $P2$ is a single clock process, according to the requirement $P2$ is synchronizing with the clock $ClockD$.

2. At the starting of every clock cycle (just after synchronization), $P2$ will send a valid/invalid data by concurrently communicating with the **Receiver**. The communication takes place with the following conditions:

   - The $P2$ will send a valid data through the channel $j.dOut$ if it has not received any stall request from the **Receiver**, during the previous cycle. $P2$ cannot send any valid data irrespective to the stall request if x is empty. The *sender token* will decide which valid data from x to be send.
   - The empty condition is depends on both ne and oe. If $P2$ receives 1 through both the channels then it means that the x is empty.
   - If $P2$ cannot send any valid data then it will send an invalid data by engaging in the event $j.void$, simultaneously with its **Receiver**.
   - After sending a valid data the *sender token* must be update.
   - The status of x may change after sending a valid data, hence $P2$ engages in $ufill$ simultaneously with the full/empty detectors after sending a valid data.

   All this work must be done within one clock cycle of $ClockD$ and after this, $P2$ will again synchronize with $ClockD$ and starts the execution of the next cycle.

   Full Detector The *full detector* is an asynchronous process, means not synchronized with any clock. Its purpose is to output the most recent status (full

or not full) of x. The *full detectors* knows the $capacity$ of x and also the number of current available items in x. By taking a $fill$ input, it will increment its size by one and by taking a $ufill$ input, it will decrement its size by one. It is always ready to take an input through the channel $fill$ or $ufill$. It outputs the status of x by comparing the $capacity$ of x with the total available items in x and it also always ready for the output. Since we are using *2 flip-flop* synchronizer, the full detector will send status as full, if the total available items in x is greater than $capacity - 2$. The reason is explained in section 5.2.

New Empty Detector The *new empty detector* is also an asynchronous process. It purpose is to output the most recent status (empty or not empty) of x. It almost similar with full detector, the only difference is that it outputs the status as empty if the total available items in x is less than 2. The detail explanation of *new empty detector* is already explained in 5.2.

Original Empty Detector It is almost same as *new empty detector*, the only difference is that it will output empty only when x will become completely empty.

SyncR It is a synchronizer on the receiver side of the MRS. It is a single clock process and its purpose is to synchronize the output of the asynchronous *full detector*. It has the same clock as the $P1$ has. Internally, it consists of two flip-flops. Hence any information passing through the synchronizer will take two cycles. The output $i.stall$ of this synchronizer will go directly to the **Sender**. The **Sender** will take this input and depending on this the **Sender** will send valid/invalid data in the next cycle.

SyncS It is a synchronizer used in the sender side of the MRS. It is a single clock process and belongs to the same clock as the $P2$ has. Its purpose is to synchronize the output of the *new empty detector*. The output of this synchronizer ne, is receive by $P2$.

SyncM This synchronizer is used to synchronize the output of the *original empty detector*.It is also a single clock process and belongs to the same to the same clock as $P2$ has. This synchronizer is not exactly same as the other two synchronizer. And its nature is clearly understandable from explanation of *original empty detector* in section 5.2.

5.4.3 The CSP Model

Now we will give the complete CSP model of the MRS. As in the previous section we already explained all the processes that we used in our model, it will be very easy to understand the model. We have not mentioned any thing about the alphabets of each process separately. All the external events of any process is shown in the Figure 5.5. For example, the process $P1$ has external events as $i.dIn, i.void, fill$ and $tockR$. The process $P1$ may have some other events but all these will be internal events. Similarly, the *full detector* has three external events $fill, ufill$ and af. We have written and explained each subprocess of the MRS, one-by-one and the complete model of MRS is given at the end.

5.4.3.1 Clocks:

As we already mentioned in section 5.3.1 that we have an array of clocks named $Clocks$ and all clocks belongs to this array. The CSP model of the ith clock in $Clocks$ is

$$Clocks[i] = tock_i \overset{d_i}{\Rightarrow} Clocks[i].$$

The details explanation is given in section 5.3.1.

5.4.3.2 P1:

The CSP model of $P1$ is

$$P1_{(rtock)} = tockR \rightarrow IN1; P1_{(rtock)}$$

$$\begin{aligned} IN1 = \; & iVoid? \rightarrow SKIP \\ & \Box \\ & iDIn?x[rtock] \rightarrow (rtock + +\%n|||fill!) \end{aligned}$$

Here the process $P1$ is synchronizing with a clock $ClockR$. Whenever we use this $P1$, we replace its tock event so that it can synchronize with a clock from the array $Clocks$. This can be done by event renaming procedure of CSP. In the above model $P1$ first waits for the event $tockR$, because the event $tockR$ can only occur when $ClockR$ also agree to engage in it. Note that we have not shown any concurrency of $P1$ with $ClockR$ here. Later when we will define the model for the whole MRS, we will show the concurrency.

After synchronizing with $ClockR$ a new cycle begins and an subprocess $IN1$ will start immediately. The purpose of $IN1$ is to read either data or void values on the input port. It will read a data if the sender sends a valid data otherwise it will read a void. If $IN1$ reads a data, then it will store the data into $x[rtock]$ where $rtock$ is the *receiver token*. After storing the data it will concurrently send a $fill$ request as well as update the *receiver token*. The output $fill$ must concurrently engage with the full/empty detectors which we define later. For now we can assume that the output $fill$ is going to an asynchronous environment, which is always ready to input it. After these operations, the subprocess $IN1$ will terminate successfully. At tis moment the process $P1$ finished all its task for this clock cycle, hence it will again return to its initial state and waits for the next $tockR$ event of $ClockR$.

5.4.3.3 P2:

The process $P2$ has four state variables namely $stl, eptN$,$eptO$ and $stock$. stl is used to store the stall request value received during the previous clock cycle. $eptN$ and $eptO$ stores the new empty status and the original empty status received during the previous clock cycle. $stock$ is used as *sender token*. The CSP model of $P2$ is,

$$\begin{aligned} P2_{(stl,eptN,eptO,stock)} \;=\; tockD \rightarrow\; & (IN2|||OUT; P2 \\ & \lhd\, stl \neq 0 \&\& (eptN \neq 0 || eptO \neq 0) \,\rhd \\ & (IN2|||jVoid! \rightarrow SKIP); P2 \\ &) \end{aligned}$$

$$IN2 = (jStall?stl|||ne?eptN|||oe?eptO).$$

$$OUT = jDout!x[stock] \rightarrow (stock + +\%n|||ufill!)$$

Here the process $P2$ is synchronizing with a clock $ClockD$. Whenever we use this $P2$, we replace its tock event in such a way that it can synchronize with a clock from the array $Clocks$. This can be done by the event renaming procedure of CSP. The process $P2$ will wait for the $tockD$ event to occur in $ClockD$ and then starts its normal execution. Note that we have not shown any concurrency of $P2$ with $ClockD$ here. Later when we will define the model for the whole MRS, we will show the concurrency.

After synchronizing with $ClockD$, a new cycle begins. Now immediately two subprocess can start depending on the given condition. If no stall request has been

received during the previous clock cycle and at least one empty signal received during the previous clock cycle is 0, then the subprocess $IN2|||OUT; P2$ will start immediately, otherwise the subprocess $(IN2|||j.void! \rightarrow SKIP); P2$ will start.

Subprocess $IN2|||OUT; P2$ **:** In this process the two subprocess $IN2$ and OUT will execute concurrently in interleaved manner and after successful termination of both, the process will return to execute $P2$ again. Means all tasks for this clock cycle finished and $P2$ will again wait for the starting of the next clock cycle. Now the purpose of $IN2$ is to simultaneously take the inputs through the following channels,

- $j.stall$– The **Receiver** will send stall request through this channel. It will send 0 for no stall request and 1 for stall request.
- ne– The output of the *new empty detector* is synchronized by a synchronizer and then send to $P2$ through this channel.
- oe– The output of the *original empty detector* is synchronized by a synchronizer and then send to $P2$ through this channel.

The value received through this channels $j.stall, ne$ and oe are stored in the state variables $stl, eptN$ and $eptO$ respectively. This value will be used in the next clock cycle to decide which subprocess should start.

The purpose of OUT is to output a valid data from $x[stock]$, where $stock$ is the *sender token*. After sending the data it will concurrently send a $ufill$ request as well as update the *sender token*. The output $ufill$ must concurrently engage with the full/empty detectors which we define later. For now we can assume that the output $ufill$ is going to an asynchronous environment, which is always ready to input it.

After successful termination of $IN2|||OUT$ the process will again return to $P2$ and wait for the next clock tick.

Subprocess $(IN2|||jVoid! \rightarrow SKIP); P2$**:** In this process the subprocess $IN2$ and $j.void! \rightarrow SKIP$ will execute concurrently in interleaved manner and after successful termination of both, the process will again return to $P2$ and wait for the next clock tick. The purpose of $IN2$ is already explained above. The subprocess $j.void! \rightarrow SKIP$ is just sending a void (invalid data) through the channel $j.void$ and terminates successfully.

5.4.3.4 Full Detector:

As we already mentioned that *full detector* is an asynchronous block. Hence when we model it in CSP we assume that it always ready to take input and also also always ready to send output. The $FULL$ process has two state variables $size$ and cap. $size$ is used to count the number of currently available items in x and cap indicates the maximum capacity of x.

$$FULL_{(size,cap)} = INFULL|||OUTFULL$$

$$\begin{aligned} INFULL &= fill \Rightarrow size++ \Rightarrow INFULL \\ &\quad \Box \\ &\quad ufill \Rightarrow size-- \Rightarrow INFULL \end{aligned}$$

$$\begin{aligned} OUTFULL &= af!1 \Rightarrow OUTFULL \\ &\quad \lhd size + 2 > cap \rhd \\ &\quad af!0 \Rightarrow OUTFULL \end{aligned}$$

The main $FULL$ process is divided into two concurrent (interleaved) subprocess $INFULL$ and $OUTFULL$. $INFULL$ is responsible for taking input coming from $P1$ or $P2$ (see black box diagram in Figure 5.5) and also changing the status according to the new input. $OUTFULL$ is responsible for sending the current status to the synchronizer whenever the synchronizer needs it. That is the reason for which $FULL$ should always ready to send output.

Subprocess $INFULL$: The process has two choice at the starting of its execution. If $fill$ comes first then it will start executing the upper part of the choice operator and if $ufill$ comes first then it will execute the lower part. Note that we have mentioned that $FULL$ is always ready to take input but one can argue about its correctness, if one input ($fill$ or $ufill$) come during the execution of $INFULL$ for the previous input. In this case our clarification is that CSP consideres each action as atomic, means they execute instantly. Hence after choosing an input ($fill$ or $ufill$) the remaining execution will finish instantly.

Subprocess $OUTFULL$: The process sends a 1 through the channel af if it satisfies the condition given, otherwise sends a 0. The explanation of always ready to send output is same as we explained for subprocess $INFULL$. This output is simultaneously taken by the *synchronizer* of the *full detector* (see Figure 5.5).

5.4.3.5 Empty Detector :

As we already explained that to handle deadlock in the MRS we use two types of *empty detectors* $NewEmpty$ and $OriginalEmpty$. Both $NewEmpty$ and $OriginalEmpty$ are running in interleaved manner without interacting with each other. Readers can can see the full explanation of using two empty detectors in [**?**].

$$EMPTY_{(cap)} = NewEmpty|||OriginalEmpty.$$

$$NewEmpty_{(size)} = INNE|||OUTNE.$$

$$\begin{aligned} INNE &= fill! \Rightarrow size++ \Rightarrow INNE \\ &\quad \Box \\ &\quad ufill! \Rightarrow size-- \Rightarrow INNE \end{aligned}$$

$$\begin{aligned} OUTNE &= ne!1 \Rightarrow OUTNE \\ &\quad \lhd size < 2 \rhd \\ &\quad ne!0 \Rightarrow OUTNE. \end{aligned}$$

$$OriginalEmpty_{(size)} = INOE|||OUTOE.$$

$$\begin{aligned} INOE &= fill! \Rightarrow size++ \Rightarrow INOE \\ &\quad \Box \\ &\quad ufill! \Rightarrow size-- \Rightarrow INOE \end{aligned}$$

$$\begin{aligned} OUTOE &= oe!1 \Rightarrow OUTOE \\ &\quad \lhd size = 0 \rhd \\ &\quad oe!0 \Rightarrow OUTOE. \end{aligned}$$

The two subprocess $NewEmpty$ and $OriginalEmpty$ are asynchronous processes and their operations are almost same as Full detectors. The only difference is in the comparison operator. So no need to explain them here.

5.4.3.6 $SyncN$:

The name $SyncN$ means normal synchronizer. Here we assume that $SyncN$ is synchronizing with a general clock whose tock event is $tock$. Later when we use it for a particular clock we will change its tock event by event renaming. For now assume that this is a synchronizer, which can operate under any clock by just changing its $tock$ event. The black box diagram of this synchronizer is given in figure 5.6.

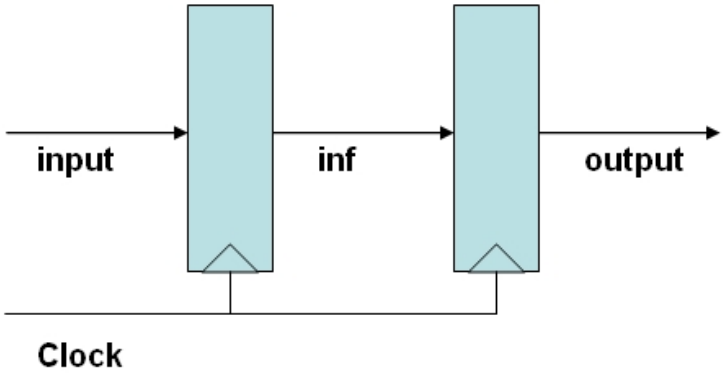

Figure 5.6: *The black box diagram for $syncN$.*

$$SyncN = D1|||D2$$

$$D1_{(a,b)} = tock \Rightarrow (input?a|||inf!b); D1_{(a,a)}$$

$$D2_{(a,b)} = tock \Rightarrow (inf?a|||output!b); D2_{(a,a)}$$

$SyncN$ is a combination of two subprocesses $D1$ and $D2$ running in concurrent with each other. At each clock cycle $D1$ will take a input from its $input$ channel and at the same time sends an output through inf. The output send by $D1$ is actually the input received in the previous cycle. At the same clock cycle $D2$ will simultaneously input the value in inf sending by $D1$. During this input, $D2$ will also simultaneously output the value it received from $D1$ in the previous cycle. $D2$ outputs through a channel $output$.

5.4.3.7 $SyncM$:

The name $SyncM$ means modified synchronizer. This synchronizer can also able to operate on any clock by just changing the $tock$ event by event renaming. The process has two concurrent subprocesses $D3$ and $D4$. The operation of $D3$ is same as $D1$ and $D2$. But the subprocess $D4$ is different from them.

$$SyncM = D3||D4$$

$$D3_{(a,b)} = tock \Rightarrow (aoe?a|||inf!b); D3_{(a,a)}$$

$$\begin{aligned} D4_{(a,b,stl,eptN)} \quad = \quad & tock \Rightarrow (inf?a|||jStall?stl|||en?eptN|||oe!b); \\ & (D4_{(a,1,stl,eptN)} \\ & \lhd stl \neq 0 \&\& (eptN \neq 0 || y \neq 0) \rhd \\ & D4_{(a,a,stl,eptN)} \end{aligned}$$

Subprocess $D4$**:** At each clock cycle $D4$ simultaneously do the following operations:

- Input the value send by $D3$ through the channel inf and store it in a.
- Input the stall request send by the **Receiver** and store it in stl.
- Input the output of *new empty detector* (after passing through the synchronizer) and store it in $eptN$.
- Output the value of b through oe. The value b is decided in the previous cycle according to the given condition. The details explanation about this synchronizer is already given in section 5.2.3.2.

5.4.3.8 Event Renaming

As we already explained in section **??** that it is possible to reuse of particular descriptions of process behavior without the need to rewrite the process in full by just replacing all the original event names with the new ones. For example, consider P be a function having alphabet $(a, b, \surd)$ and f be a function which maps the alphabet of P onto a set of symbols A,

$$f : \alpha P \rightarrow A$$

Now we define the process $f(P)$ as one which engages in the event $f(a)$, whenever P would have engaged in a and this is same for all other events also. Hence,

$$\alpha f(P) = f(\alpha P)$$

Renaming $P1$ **:** Consider $f_i : \alpha P1 \rightarrow A_i; i \in [1, N]$ be a set of functions which maps each event of $P1$ onto a new set of events A_i, such that,

$$f_i(tockR) = tock_i,$$
$$f_i(y) = y, \quad \text{if } y \text{ is not } tockR.$$

By renaming $P1$ in this way, it is possible to synchronize $P1$ with any clock from the array $Clocks$ (see section 5.3.1). Here i means the ith clock in the array $Clocks$.

Renaiming $P2$ **:** Consider $g_i : \alpha P2 \rightarrow B_i; i \in [1, N]$ be a set of functions which maps each event of $P2$ onto a new set of events B_i, such that,

$$g_i(tockD) = tock_i,$$
$$g_i(y) = y, \quad \text{if } y \text{ is not } tockD.$$

By renaming $P2$ in this way, it is possible to synchronize $P2$ with any clock from the array $Clocks$ (see section 5.3.1). Here i means the ith clock in the array $Clocks$.

Renaming $SyncN$ **:** We need to rename $SyncN$ by two functions, one for *full detector* and one for *empty detector*.

- Consider $h_i : \alpha SyncN \rightarrow C_i; i \in [1, N]$ be a set of functions which maps each event of $SyncN$ onto a new set of events C_i, such that,

  $$h_i(tock) = tock_i,$$
  $$h_i(input) = af$$
  $$h_i(output) = i.stall$$
  $$f_i(y) = y, \quad \text{if } y \notin (tock, input, output).$$

  Now the new processes $h_i(SyncN); i \in [1, N]$ can be used as a synchronizer for the full detector.

- Consider $k_i : \alpha SyncN \rightarrow D_i; i \in [1, N]$ be a set of functions which maps each event of $SyncN$ onto a new set of events D_i, such that,

  $$k_i(tock) = tock_i,$$
  $$k_i(input) = ane$$
  $$k_i(output) = ne$$
  $$f_i(y) = y, \quad \text{if } y \notin (tock, input, output).$$

  Now the new processes $k_i(SyncN); i \in [1, N]$ can be used as a synchronizer for the new empty detector.

Renaming $SyncM$ **:** Consider $m_i : \alpha SyncM \rightarrow E_i; i \in [1, N]$ be a set of functions which maps each event of $SyncM$ onto a new set of events E_i, such that,

$$m_i(tock) = tock_i,$$
$$m_i(y) = y, \quad \text{If } y \text{ is not } tock.$$

Now the new processes $m_i(SyncM); i \in [1, N]$ can be used as a synchronizer for the original empty detector.

5.4.3.9 The process MRS:

Now we are ready to define the CSP model for MRS. We define it based on the concepts we explained in the section 5.4.2 and 5.4.3. Reader can also refer to the figure 5.5 for understanding the model. The model is,

$$\begin{aligned} MRS_{(x,i,j)} = \quad & f_i(P1)||g_j(P2)||h_i(SyncN)||k_j(SyncN)|| \\ & m_j(SyncM)||FULL||EMPTY||Clocks \end{aligned}$$

where x is the array used to store data, i means the receiver side of the MRS is belongs to the ith clock of $Clocks$ and j means the sender side of the MRS is belongs to the jth clock of $Clocks$.

5.5 Analysis

This section defines the behavior of multiple clock LI module and states the properties enjoyed by such a system.

5.5.1 Type of Events and Process Trace

We first define the type of events, and then, define what is meant by a process trace, its failure, and divergence.

Definition 5.5.1. *This is an input (output) event where valid data value is received (sent) by the process. An informative event of value* x_1 *on channel* a *is denoted by* $\iota^a_{x_1}$.

Definition 5.5.2. *This is an input (output) event when valid data are not received (sent), and hence, the void channel receives (delivers) a signal from (to) the sender (receiver) process. Such events are termed as void events. Such an event on channel a is denoted by ϕ^a. When a ϕ event occurs, there is no i event and vice versa.*

Definition 5.5.3. *In addition to data and voids, the process can send stall events to its sender to prevent it from sending more data items. Such an event on channel a is denoted by σ^a.*

Definition 5.5.4. *The trace of the behavior of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. Since a process can either single clock or multiple clock, the engaging time is considered as the time of real clock (global watch). The set of all traces of a given process is denoted by $\mathfrak{T}[P]$.*

Note that if we only consider the single clock processes then the definition of trace given here is same as the definition of trace given in [1]. But for multiple clock processes, the trace is the interleaving of the traces of its all single clock subprocesses. Below we have given example for both type of traces.

Example 5.1. Consider a single clock process with input ports a and c and output port b. One of its trace is

$$tock, \iota^a_{x_1}, \iota^c_{w_1}, \iota^b_{y_0}, tock, \iota^a_{x_2}, \iota^c_{w_2}, \iota^b_{y_1}, tock, \iota^a_{x_3}, \phi^c, \iota^b_{y_2}, \sigma^b, \phi^a, \iota^c_{w_3}, \phi^b, \sigma^b, \ldots$$

where the output on channel b is a function of the inputs received on channels a and c, i.e., $y_i = f(x_i, w_i)$. Note that the process receives void data on c and stall request on b during the third clock cycle. Another importing thing here is, we are considering b as a port and not a channel, it is actually a combination of channels. This is the reason that b can do both output (data) and input (stall request).

Example 5.2. This example shows that void and stall events on certain channels of a single clock process lead to void and stall events on other channels in future clock cycles. But we need to insert only one set of continuous void or stall events on other channels. Means, if it is required to insert void or stall events on the other channels then we insert them one after another without any other events in between them. This phenomenon is similar to the procrastination effect described in [2] and

also to the multiple clock procrastination effect when $m = 1$[cf. definition 3.4.6]. For simplicity, we will only consider example process with one input port a and one output port b. A sample trace is given below:

$$T_1 = \langle tock, \iota^a_{x_1}, \iota^b_{y_0}, tock, \iota^a_{x_2}, \iota^b_{y_1}, tock, \phi^a, \iota^b_{y_2}, tock, \phi^a, \phi^b, tock, \iota^a_{x_3}, \phi^b, tock, \iota^a_{x_4}, \iota^b_{y_3} \rangle$$

Note that the void event in third clock cycle resulted in a void event in forth cycle.

Example 5.3. Consider a multiple clock process P having two single clock subprocess $P1$ and $P2$. $P1$ is synchronizing with clock C_1 having clock period 20 and $P2$ is synchronizing with clock C_2 having clock period 16. The purpose of $P1$ is receive some input data from the environment through the input port a and store them in a shared memory M. And the purpose of $P2$ is to send the data from the shared memory M to the environment through channel b. Note that here we are not considering the different issues of using shared memories. Now the simple rule is that $P2$ cannot send any data, which $P1$ has not been received yet. Since the two process are synchronizing with two different clocks hence there clock cycles will overlap with each other. Due to this reason we have shown a possible trace of P in the figure 5.7(a). If we write this in general form then it will be,

$$\begin{aligned} T_2 = \; & \langle tock_1, \iota^a_{i_0}, tock_2, \iota^b_{i_0}, tock_1, \iota^a_{i_1}, tock_2, \iota^b_{i_1}, tock_2, \phi^b, tock_1, \iota^a_{i_2} tock_2, \iota^b_{i_2}, tock_1, \\ & \iota^a_{i_3}, tock_2, \iota^b_{i_3}, tock_1, \phi^a, tock_2, \phi^b, tock_1, \iota^a_{i_4}, tock_2, \iota^b_{i_a}, tock_2, \phi^b, tock_1, \iota^a_{i_5}, \\ & tock_2, \iota^b_{i_5}, tock_1, \phi^a, tock_2, \phi^b, tock_1, \iota^a_{i_6}, tock_2, \phi^b \rangle \end{aligned}$$

Example 5.4. This example shows that void and stall events on certain channels of a multiple clock process lead to void and stall events on other channels in future clock cycles. But here inserting only one-time-continuous void events may not be sufficient. For example consider the process P mentioned in the previous example. If $P1$ inputs void at the 0th clock tick (assume clock ticks start from 0) then $P2$ cannot send data at its 0th clock tick. So in this case $P2$ bound to send one void at this cycle. Now $P1$ receives a data at its 1st tick hence $P2$ can send the data at his 1st tick. But $P2$ cannot send any data at its 2nd tick because, $P2$'s 2nd tick is occurring before the 2nd tick of $P1$. Hence $P2$ bound to send a void at its 2nd clock cycle. If we compare this trace with the trace T_2 we can see that in T_2, subprocess $P2$ sends i_0 and i_1 in two consecutive ticks (0th and 1st), but here it is not possible, because of inserting a stall at the 0th tick of $P1$. The problem is showing the figure 5.7(b) with dotted

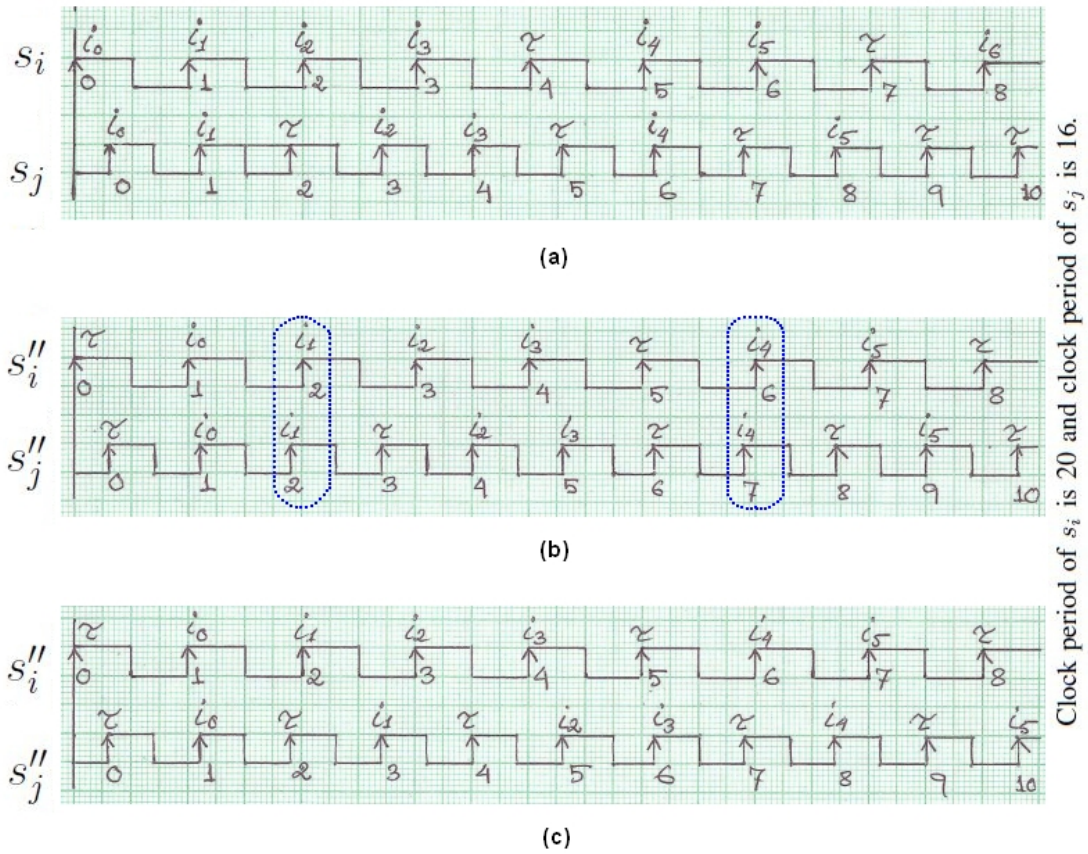

Figure 5.7: *The example of trace as well as* $\mathcal{MPE}$.

eclipse. So its means that for inserting a void event on a certain channel we may need to insert void events on any channels at any ticks in future. This phenomenon is exactly similar to our Multiple Clock Procrastination Effect $\mathcal{MPE}$, defined in section 3.4. The resultant trace is shown graphically in the figure 5.7(c).

The latency insensitive processes synchronize with their neighbors to transfer information. The synchronizations we consider are blocking (i.e., communication on nonbuffered channels), and hence, the system may deadlock if a process is not able to engage in such a synchronized event. This case, however, does not arise as the receiver is always ready to accept any type ($\iota/\phi/\sigma$) of signal sent to it. This is inherent in the construction of the LI process.

5.5.2 Latency Equivalence

As seen earlier, void and stall on certain channels lead to void and stall events on other channels. We therefore cannot compare the traces directly. Instead, we use the traces to derive events along individual channels. This helps in establishing latency equivalence.

Definition 5.5.5. *A stream (S_a^T) is a sequence of events occurring in trace T, along a given channel (a) separated by the tock.*

Example 5.5. For example, the stream on channel a for trace T_1 in the above example is

$$S_a^T = \langle tock, \iota_{x_1}^a, tock, \iota_{x_2}^a, tock, \phi^a, tock, \phi^a, tock, \iota_{x_3}^a, tock, \iota_{x_4}^a, \ldots \rangle$$

Definition 5.5.6. *Restricting the stream to informative events gives an informative stream. For a stream S, its informative stream is given by $S_I = S \uparrow \iota$.*

Example 5.6. For the stream in example 5.5, the informative stream is

$$S_{aI} = \langle \iota_{x_1}^a, \iota_{x_2}^a, \iota_{x_3}^a, \iota_{x_4}^a, \ldots \rangle$$

Definition 5.5.7. *Two streams are said to be latency equivalent (denoted $\equiv_\tau$) if they have the same informative streams. For streams S_1 and S_2*

$$(S_1 \equiv_\tau S_2) if (S_{1I} = S_{2I}).$$

Example 5.7. Consider the two streams S_1 and S_2 on channel a and b respectively fro the trace T_2 given in the example 5.3 [see Figure 5.7(a), for graphical version]:

$$S_1 = \langle tock_1, \iota_{i_0}^a, tock_1, \iota_{i_1}^a, tock_1, \iota_{i_2}^a, tock_1, \iota_{i_3}^a, tock_1, \phi^a, tock_1, \iota_{i_4}^a, tock_1, \iota_{i_5}^a, tock_1, \phi^a, tock_1, \iota_{i_6}^a, \ldots \rangle$$

$$S_2 = \langle tock_2, \iota_{i_0}^b, tock_2, \iota_{i_1}^b, tock_2, \phi^b, tock_2, \iota_{i_2}^b, tock_2, \iota_{i_3}^b, tock_2, \phi^b, tock_2, \iota_{i_4}^b, tock_2, \phi^b, tock_2, \iota_{i_5}^b, tock_2, \phi^b, tock_2, \phi^b, tock_2, \iota_{i_6}^b, \ldots \rangle$$

Their respective informative streams are:

$$S_{1I} = \langle \iota_{i_0}^a, \iota_{i_1}^a, \iota_{i_2}^a, \iota_{i_3}^a, \iota_{i_4}^a, \iota_{i_5}^a, \iota_{i_6}^a \rangle$$

$$S_{2I} = \langle \iota_{i_0}^b, \iota_{i_1}^b, \iota_{i_2}^b, \iota_{i_3}^b, \iota_{i_4}^b, \iota_{i_5}^b, \iota_{i_6}^b, \rangle$$

These streams are same, and hence, the original streams are latency equivalent.

Definition 5.5.8. *Consider two latency insensitive processes (single clock or multiple clock) P and Q having the same alphabet (except tock event). These two processes are latency-equivalent (denoted $P \equiv_\tau Q$) if the following condintions are satisfied for all traces of P and Q:*

1. *Input streams sent to all the input ports of P are latency-equivalent to the inputs streams sent to the corresponding input ports of Q.*
2. *Output streams generated by P on all its output ports are latency-equivalent to the corresponding streams produced by Q.*
3. *The pairs of equivalent input and output streams belong to the same trace.*

If the input data streams are latency-equivalent, then the two processes will receive similar data sets in the same order. The results will therefore be computed using the same data sets. As per Lemmas (5.1, 5.2, and 5.3) proved in [1] and the Lemma 5.6.1 (proved later), data are not lost and results are computed using all values. The results generated by both the processes will therefore be identical. These will then be sent to the receiving processes depending on the stall requests. Given that the output streams are also latency equivalent and the results are generated using correct inputs, we can say that the two processes P and Q are latency-equivalent.

Note that when we are talking about latency-equivalence, we are not considering the clocks. In the example 5.7, the two streams S_1 and S_2 belongs to two different clocks but still we are saying them as latency-equivalent. Hence, latency-equivalence is only depend on the ordering of the informative events. It has no relation with the actual occurring-time of the informatives events.

Definition 5.5.9 (Multiple Clock LI Processes)**.** *Consider P is a multiple clock process and it has some properties. Now if it is possible to insert arbitrary number of delays in any channel of P without violating the properties of P, then P is called a Multiple Clock LI process (same as multiple clock patient process defined in section 3.4.1).*

5.6 Properties

As the model is tolerant to delays on input channels, we need to check if it provides safety against data overwriting. Following properties also gives a better insight into

the working of an MCLI process. The properties will also prove that the MRS is an MCLI process.

MRS belongs to two clocks, receiver clock and sender clock. The receiver side of MRS belongs to receiver clock and the sender side of MRS belongs to sender clock. Hence, the **Sender** module which sends data to the MRS must belongs to the same clock as the receiver side of MRS has. Also the **Receiver** module which receives data form MRS must belongs to the same clock as the sender part of MRS has. This is mandatory because in CSP module must synchronize for the input/output.

Property 5.6.1. *The* **Sender** *module (module which send data to MRS) and the receiving part of MRS must have same clock.*

Property 5.6.2. *The* **Receiver** *module (module which receive data from MRS) and the sending part of MRS must have same clock.*

The order of inputs on a particular channel is important between two consecutive cycles.

Property 5.6.3. *Between two consecutive transition on the same channel, there is a tock event.*

As we can see from the model of MRS that each input/output channels are synchronizing with a tock event at the starting of every clock cycle (may be different channel synchronize with different clocks). Hence the above property is proved. Note that here input/output channels means the channels input to the MRS from the environment and the channels output from the MRS to the environment.

Property 5.6.4. *Infinitely many actions do not happen between two ticks of same clock.*

This is a safety property to check that no single clock subprocess of the MRS diverge. It guarantees that there are finite number of events occuring between consecutive $tock$ event of same clock. Multiple clock process is just an interleaving of single clock subprocesses. Hence if no subprocess diverge then the MRS will not diverge too. This property also helps to satisfy the assumption of completing of computation by each single clock subprocess within its one clock cycle.

Property 5.6.5. *A $stall$ event results a void event.*

Proof: A $stall$ event means a stall request came from the **Receiver**. Now if the request came at nth cycle (the clock belongs to the receiver side clock of MRS) then at the next cycle $[(n+1)th]$, the MRS will send a void output.

From the proof of the above property we get a new property.

Property 5.6.6. *Stalls are acted upon one clock cycle (sender side clock) later.*

Property 5.6.7. *Stalls may not always ripple through a pipeline of modules (MRS, LI Computational Bloks and LI Connectors).*

Proof: Since the sender side and the receiver side of the MRS are not belongs to same clock. Let the receiver side clock is C_1 and the sender side is C_2. Consider that during two ticks of C_1, C_2 ticks four times. Also consider that the 1st tick of both C_1 and C_2 occurs simultaneously and the sender side receives a stall request from **Receiver** at its 1st tick. Now, due to this stall request, sender side cannot send any data at its 2nd tick but if no more stall request arrives at its 2nd ticks than it can send the data at its 3rd tick. Since 3rd tick of C_2 will occur before the 2nd ticks of C_1, hence the stall request will no longer need to pass to the **Sender**. But if the relation between C_1 and C_2 is just opposite as that we mentioned above then it may need to send more than one stall request to the **Sender**, for a single stall request from the **Receiver**.

The receiver part $P1$ of MRS, accesses the global array x, according to the receiver token $rtock$. Receiver token $rtock$ is rotating clock-wise through the array. If $trock = 1$, then at the next cycle of $P1$, it will write the input data into $x[1]$ and move the $rtock$, 1 position to the right, provided there is a valid data to write. But if there is no valid data then the $rtock$ will remain in same position until a valid data stores at this position of x. Similarly, the sender part $P2$ of MRS, accesses the global array x, according to the sender token $stock$. Sender token $stock$ is also rotating clock wise through the array. If $stock = 0$, then at the next cycle of $P2$, it can send $x[0]$ to the **Recever** and move the $stock$ 1 position to the right, provided x is not empty and current cycle has not received any stall request. But if the x is empty or $P2$ receives a stall request in its current cycle then $P2$ will not going to access x in its next cycle. Also we assume that initially, both $rtock$ and $stock$ will be equal to 0.

Property 5.6.8. *Both sender $P1$ and receiver $P2$ will not access same item of x (array of cell according to the block diagram given in Figure 5.1) simultaneously.*

Proof: Both $P1$ and $P2$ accessing the same item of x means $rtock = stock$. From the above discussion we can say that $rtock$ will become equal to $stock$ when x will be empty (not necessarily only at the initial state). But we already mentioned that $P2$ will not access x, if it is empty. Once x become empty and $rtock$ become equal to $stock$, $P1$ will first write a data on $x[rtock]$ and then empty detector will change the status of x from empty to non-empty. Now $P2$ can access the data but in the mean time $rtock$ is already shifted right by at least 1 position.

Property 5.6.9. *The sender $P2$ will send data in the same order as $P1$ is receiving them.*

Proof: Since both sender-token and receiver-token are rotating clock-wise and both are not taking any illegal jumps, hence definitely the order will maintain.

Lemma 5.6.1. *The data in the global array x will not lost or overwrite.*

Proof: The data in x can only be lost if $P1$ overwrite another data in the same position before $P2$ can able to send it. Now we know that $P1$ writes data only at $x[rtock]$ and $rtock$ moves clock-wise. Hence after writing a data in a position of x, $P1$ can again write on the same position, after a complete circular (clock-wise) rotation of $rtock$. But if at that time the data will still there then it means the x is full. In such a case the **Sender** will not send any valid data until x becomes not-full. Hence, there is no chance of overwriting.

Property 5.6.10 (Liveness-1)**.** *Data received on input will eventually be output.*

Proof: Clearly if the **Receiver** sends infinite number of stall requests then it will not possible for MRS to output data that it has already received from the input channel. But as we know that the **Receiver** is either an LI Computational Block or an LI Connector (defined in [1]). Since the author of [1] already proved that LI Computational Blocks and LI Connectors cannot send infinite number of stalls hence, data received on input will eventually be output.

Property 5.6.11 (Liveness-2)**.** *MRS does not send infinite number of stall events to its* **Sender**.

Proof: MRS will send stall events to its **Sender** if the data array x become full. And MRS will send infinite number of stall events to its **Sender** if x remains full.

Now x remains full means MRS is not outputting the data that it already received. But according to Property 5.6.10 data received by MRS will eventually be output. Hence there is no need to send infinite stall request to the **Sender**.

5.7 Multiple Clock LI Systems (MCLI)

So far we have defined and analyzed an MCLI process MRS but we have not said any thing about MCLI sytems. As we mentioned in section 5.4.1 that MCLI systems consists of MRS, LI Combinational Blocks and LI Connectors. The author of [1] only proves that the LI Computational Blocks and the LI Connectors are LI Processes. Hence first of all we have prove that these two are also MCLI processes.

Theorem 5.7.1. *An LI process (single clock patient process [2]) is also an MCLI process (multiple clock patient process 3.4.1).*

Proof: We know that both LI and MCLI processes can handle any arbitrary number of delays on their channels, without violating their properties. So the only difference between LI processes and MCLI processes is the way they handle the delays. But if we see the definition of Multiple Clock Procrastination Effect $(\mathcal{MPE})$, from section 3.4, we can observe that the way an MCLI process handles delay when there is only one clock, is exactly same as the way a single clock LI process handles delay. Hence it is proved that single clock LI processes are also MCLI process.

5.7.0.1 Small Change in LI Computational Blocks and LI Connectors [1]

In this section we explain the small change we have done in LI Computational Blocks and LI Connectors without violating any properties of them. The change is very simple.

Change in LI Computational Blocks (LICB) : As mentioned in [1], at every clock cycle, the LICB sends a $stall!1$ event to the **Senders** whose corresponding port in LICB have unused data. But LICB is not sending any thing to those **Senders** whose corresponding port in LICB have no unused data. We are just adding that the LICB will now send a $stall!0$ event to those **Senders** whose corresponding port in LICB have no unused data. This change will not violate any property for LICB as well as LI Connectors. Even this is not needed if the **Sender** is also an LICB or an LI Connectors. But this change is necessary if the **Sender** is an MRS.

They can be done by just changing the two subprocesses $STLOUT$ and $DTOUT$ as follows,

$$STLOUT = (|||_{j\in\mathcal{B}} \quad j.void!) \rightarrow (|||_{i\in\mathcal{A}}(i.stall!1 \lhd xnw[i] \neq 0 \rhd i.stall!0))$$

and

$$\begin{aligned} DTOUT \quad = \quad & (|||_{j\in\mathcal{B}} \quad j.dOut!y[j]) \rightarrow (|||_{i\in\mathcal{A}} i.stall!0) \\ & \lhd ynw = 1 \rhd \\ & ((|||_{j\in\mathcal{B}} \quad j.void!) \rightarrow (|||_{i\in\mathcal{A}}(i.stall!1 \lhd xnw[i] \neq 0 \rhd i.stall!0)) \end{aligned}$$

The readers can compare the new definition of $STLOUT$ and $DTOUT$ with there original definition proposed in [1] for LICB.

Change in LI Connectors (LICN) : The changes we done here is the same as we do it for LICB. The only difference in LICN is that it has only one **Sender** and only one **Receiver**. The two modified subprocesses $STLOUT$ and $DTOUT$ are given below:

$$STLOUT = j.void! \rightarrow (i.stall!1 \lhd xnw \neq 0 \rhd i.stall!0))$$

and

$$\begin{aligned} DTOUT \quad = \quad & (j.dOut!y \rightarrow i.stall!0) \\ & \lhd ynw = 1 \rhd \\ & (j.void! \rightarrow i.stall!0) \end{aligned}$$

The readers can compare the new definition of $STLOUT$ and $DTOUT$ with there original definition proposed in [1] for LICN.

Theorem 5.7.2. *The LI Computational Block (LICB) and LI Connectors (LICN) defines in [1] are also MCLI Processes.*

Proof: Proved according to Theorem 5.7.1.

5.7.1 MCLI System Properties

In this section we have proved the three important properties of MCLI systems.

Property 5.7.1. *The MCLI systems deadlocks if all processes reach a failure state in the same clock cycle. To verify for deadlock freedom, we must make sure that at leas one process in not in failure state.*

Proof: Assuming the environment is fair (i.e., does not send infinite number of stall and void events), the process will be capable of computing results, and hence will not be in a failure state.

Property 5.7.2. *Composing two multiple-clock latency-insensitive processes,* $MP1||MP2$*, gives a multiple-clock latency-insensitive composition.*

Proof: The main condition of the composition of two MCLI processes $MP1$ and $MP2$ is that the sender part of $MP1$ and the receiver part of $MP2$ must synchronize with same clock. But the receiver part of $MP1$ and the sender part of $MP2$ may run in different clocks. Since we already proved that a single clock LI process is also an MCLI process hence here MCLI process means both. Now parallel composition of processes results in a composition which has some external input/output channels and some internal communication channels. We will consider both cases here. The prove given here is almost same as the proof given in [1]. The only difference is that when a MCLI process receives a $stall$ event, then propagating this $stall$ events to its **Sender** is not just depends on the storage availability but also depends on the clock period (see Property 5.6.7). Now here on receiving a $stall$ event, $MP1$ selectively sends $stall$ events to its **Senders** based on the property 5.6.7.

- **External channels:** Consider the case input channels. Let $a_i(0 \leq i < n)$ be n external inputs to process MP_1 and $b_j(0 \leq j < m)$ be m external inputs to process P_2. Suppose inputs are ready on all a_i while inputs on certain b_j are slow to arrive. Consider the following situation:

  Process MP_2 receives void events on certain b_j channels.
  $\Downarrow$
  Process MP_2 sends void and stall events to processes including MP_1
  $\Downarrow$
  On receiving $stall$ request, MP_1 sends void and (selective) stall events to external processes.
  $\Downarrow$
  Sender processes connected to channels a_i are stalled.
  $\Downarrow$
  The composition is thus multiple-clock latency-insensitive on external input channels.

- **Internal channels:** These channels are those which are output of one process connected to input of the other process in the composition. Parallel composition of processes results in synchronization on these internal channels (as we already mentioned that the sender part of MP_1 and the receiver part of MP_2 are belongs to same clock). The events (i.e.,data, void, or stall) that are sent by one process will be (by definition) accepted by the other process. Hence, synchronization on an offered event is guaranteed.

  If one process of the composition is not ready for accepting further inputs, it will send stall request to the second, and therefore, the second process will send (selective) stall requests to other processes (external to the composition). Thus, the composition acts like a single process to its environment, in that the composition sends stall and void signals to the external processes if one process in the composition is stalled/slow, and is ready to accept data values when computation is feasible and/or buffer space is available. Also, as there is no data loss, the composition will be multiple-clock latency-insensitive.

5.7.1.1 A CSP model for a complete MCLI System

So far we have given the CSP model of all the modules which are used in MCLI system. But we have not given any model for a complete MCLI system, which is a composition of MRS,LICB and LICN. In this section we give a CSP model for the MCLI system shown in the Figure 5.8. The figure is almost same as the figure 3.4 given in 3.6. We just renamed some block here for our purpose. Hence for detail explanation of this figure one can read section 3.6.

The MCLI system consists of three LICB modules (namely LB_1, LB_2 and LB_3), seven LICN modules (namely LN_1, LN_2 LN_3, LN_4, LN_5, LN_6 and LN_7) and two MRS (namely MRS_1 and MRS_2). The system has two clocks $clock_1$ and $clock_2$. Let these two clocks are the 1st and 2nd clock of the array $Clocks$. Now the components belongs to $Clocks[1]$ are:

- LB_1, LN_1, LN_2, LN_7 and,
- The receiver part of both MRS_1 and MRS_2.

Similarly, the components belongs to $Clocks[2]$ are:

- $LB_2, LB_3, LN_3, LN_4, LN_5, LN_6$ and

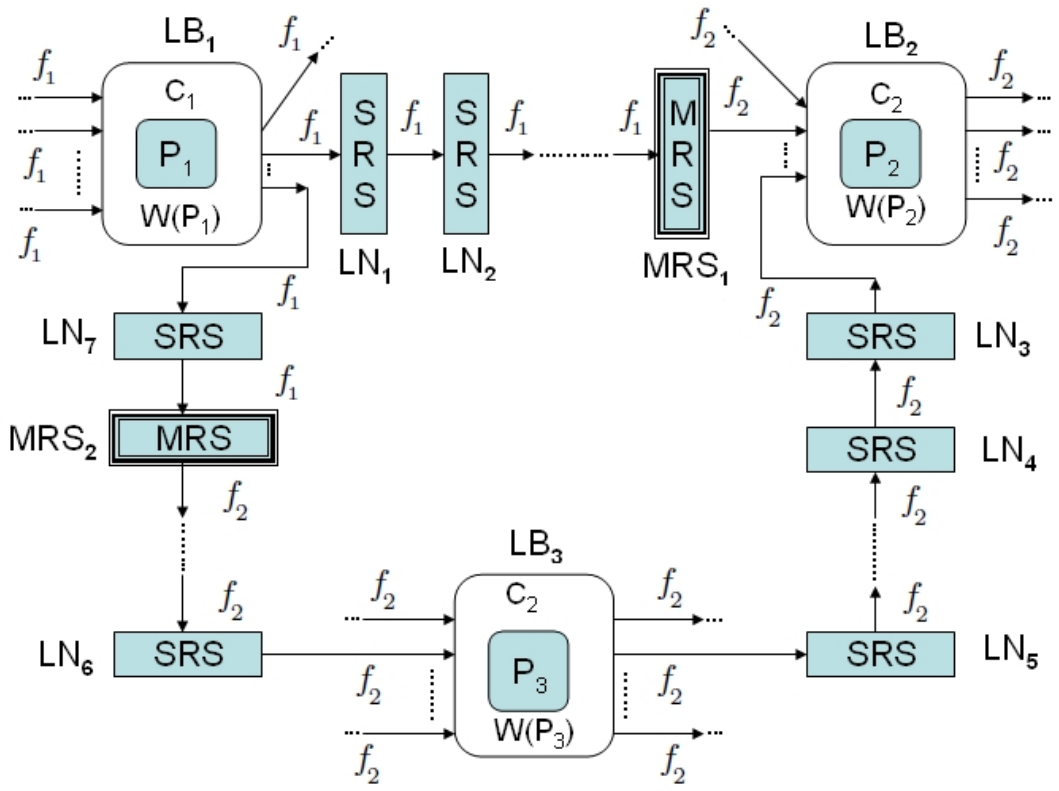

Figure 5.8: *An MCLI System.*

- The sender part of both MRS_1 and MRS_2.

One can say that the model of the complete system is just the composition of all this process. But unfortunately this will not work. The problems of such composition are mentioned below.

- $tock$ **event of LICB and LICN:** From [1] we can see that the models LICB and LICN are synchronizing with a clock whose tock event is $tock$. This clock is not in the array $Clocks$. Hence before using them we have to make them possible to synchronizewith the clocks in $Clocks$.

- **Localization Problem:** More than one process may have same channel names though they are not communicating with each other. For example the name of the input channel of all LN is $1.din$ (according to [1]). Hence they are creating problem in direct composition.

- **Synchronization Problem:** The communicating channels between two modules have different names in each module. For example, the channel communicating between LB_1 and LN_1 has a name $2.dout$ in LB_1 and $1.din$ in LN_1. For any confusion, see the CSP model of both LICB and LICN from [1].

- **Clock Composition:** If we see the model of MRS then we can see that it is composing the whole array $Clocks$ with it. Though there is no problem of writing this, a new problem will arise when more than one MRS will be present in a MCLI system. In this case the MCLI system will compose the array $Clocks$ twise (one for each MRS), which is not necessary.

Below we solve all the above problems one-by-one.

Let $r'_i; i \in [1, N]$ be the set functions which when applied to a process, maps the $tock$ event to $tock_i$. Now if we want to synchronize a process Q with the ith clock of $Clocks$, then we have to use $r'_i(Q)$ instead of Q. Note that this function can only applied to those processes, which have a $tock$ event. Since the process of LICB and LICN have the $tock$ event, we can apply this function to them to rename their $tock$ events.

Each module in a MCLI system have a unique identification number id. Let g' is a function which when applied to a process Q, maps all the external event (except the tock event) of Q into a new set, which is completely unique in this MCLI system. Here j indicates the process id. The mapping is done by concatenating the process id at the beginning of its every external event (except the tock event). In this way it is possible to make all the external events of a process in this system as unique from all other processes. Note that this function not changes the tock event of the process.

For removing the synchronization problem defined above, we use a simple technique. We change the name of each input channel of a process with the name used by the corresponding sender process for this channel. For example, consider Q and R be two processes and they need to communicate with each other through a channel. But the name of this channel in Q is a and in R it is b. Now to make the communication possible we rename the event b of R as a, which is same as the name used in Q for this channel. Let h' be the function which doing this type of event renaming. Note that at each process we are only renaming the input channels and not the output channels. Also there is no need to change the tock event, because it is renaming by

another function.

For removing the Clock Composition problem we remove the $Clocks$ from each individual MRS. Let $MRSX$ be the process which is same as MRS but the array $Clocks$ is not there as a concurrent subprocess. This removal will not create any problem, because we will use it as concurrent subprocess for the whole MCLI system.

Now the next step is change each process according to the techniques given above. If the process is an LICB or an LICN the we have to do the following modifications:

- Apply the function r' to change its $tock$ event.
- Apply the function g' to rename all the external events name as unique.
- Apply the function h' to rename all the input channels according there name in the corresponding sender process.

For example, if Q is an LICB, which need to synchronize the ith clock from $Clocks$ and also receives input from two process. Then the resultant process will be $f'_i(Q) = h'(g'(r'_i(Q)))$.

If the process is an MRS then first of all we have to remove the subprocess $Clocks$ from it. Let the new process be $MRSX$. Now the remaining process is same, but we no need the function r' here, $f''(MRS) = h'(g'(MRSX))$.

Now we are ready to define the model of the MCLI system given in the figure 5.8. It will be as follows:

$$\begin{aligned} MCLI \quad = \quad & (f'_1(LB_1)||f'_2(LB_2)||f'_2(LB_3)||f'_1(LN_1)||f'_1(LN_2)|| \\ & f'_1(LN_7)||f'_1(LN_1)||f'_2(LN_3)||f'_2(LN_4)||f'_2(LN_5)|| \\ & f'_2(LN_6)||f''(MRS_{1(x,1,2)})||f''(MRS_{2(x,1,2)})|| \\ & Clocks)\backslash \{\text{all internal events of } MCLI\} \end{aligned}$$

Here we have taken two important assumptions, they are:

- In the above model we write the name of each LICB and LICN models as LB_i and LN_i. But in there original definition [1], they are defined by P and C respectively. Writing $LB_i; i \in [1, N]$ means, we have re-written the process P, N times and each time we use a separate name for it. Same concept is valid for LICN and MRS also.

- We have not mentioned any thing about the state variables for the processes of LICB and LICN but we have mentioned it explicitly for the processes of MRS. This became necessary because we need to pass the clock indexes to the MRS (see the process MRS in the section 5.4.3.9). Though we have not mentioned any thing about the state variable of other processes we assume that they belongs to their corresponding processes.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

Latency issues in single clock domain systems have been handled in a previous work. We extended this work for connecting systems with multiple clock frequencies. The paradigm shift necessary to achieve this was demonstrated by means of examples where the original theory falls short in handling latency issues.

We defined a multiple clock LI buffer which can be used (i) in the chain of single clock relay stations; or (ii) simply as a converter, between two patient processing modules having rationally related frequencies, thus making the system LI. This buffer inputs data from one frequency domain and forwards it to another frequency domain; as well as it can tolerate back-pressure.

To handle latency issues created by input delays or back-pressure, the buffer has to insert stalls in various connecting signals. The number of stalls to be inserted depends on the ratio of the clock frequencies to be connected. The main observation being that the events in two sequences have co-relation every LCM(p_i, p_j) time units, where p_i and p_j are the clock periods of two connecting channels. The definition can be modified to take into account the setup and hold constraints of the storage elements, thus enabling implementability. It also guarantees that composing modules from the single clock and multiple clock LI design styles yield a multiple clock LI design.

6.2 Future Works

In the next semester we are going to prove our proposed theory i.e. theory of LI design for multiple clocks, with a new framework called "Process Algebra". For this we will design a process algebra model for the multiple clock LI design. Actually a model in process algebra for single clock LI design is already proposed [1]. We will extend this work for multiple clocks. below we have given a short description of process algebra.

6.2.1 Process Algebra

Process Algebra [3] provide a well studied framework for modeling and verifying concurrent systems. They all have the facility to model discrete time models, where time is considered as event. There are even extensions to process algebras which extend the timed events to multiple clocks [4][5]. The followings are some important properties of process algebra:

- They provide a concrete frame work from modeling and verification of complex and concurrent systems.
- Various tools are available for modeling process algebra equation and verify their correctness.
- Process algebra has framework which facilitates the modeling of discrete timed and mixed timed events.

6.3 Summary of this Semester Works

The summary of my works in this semester is mentioned in the Table 6.1.

Duration	Work Done
22nd May - 31st May	Learning LaTeX.
1st June - 5th June	Literature Survey.
8th June	Seminar given on single clock LI design.
9th June - 20th June	Understanding the last year work.
20th June - 31st Aug	Modifying/Extending the last year work.
1st Sep - 15th Oct	Preparing the paper "Extending the Theory of Latency Insensitive Design to Multiple Clocks". Authors: Parasara Sridhar Duggirala, Shirshendu Das and Hemangee K Kapoor.
16th Oct	Paper Submitted for IEEE JOURNAL OF TRANSACTIONS ON CAD.
17th Oct-5th Nov	Thesis preparation.

Table 6.1: *Summary of this semester works.*

References

[1] H. K. Kapoor, "A Process Algebra View of Latency Insensitive System," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 931–944, July 2009. iii, 9, 11, 90, 91, 92, 93, 94, 95, 96, 97, 110, 124, 128, 131, 132, 133, 134, 136, 137, 138, 141

[2] M. Bohr, "Interconnect scalingThe real limiter to high performance ULSI," in *Proc. IEEE Int. Electron Devices Meeting*, Dec 1995, pp. 241–244. 1

[3] R. Ho, K. Mai, H. Kapadia, and M. Horowitz, "Interconnect scaling implications for CAD," in *Proc. Int. Conf. Computer-Aided Design*, Nov 1999, pp. 425–429. 1

[4] D. Matzke, "Will physical scalability sabotage performance gains?" Sept 1997. 1

[5] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conentional microarchitectures," in *Proc. 27th Annu. Int. Symp. Computer Architecture*, June 2000, pp. 248–250. 1

[6] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Coping with Latency in SOC Design," *IEEE Micro*, vol. 22, no. 5, pp. 24–35, Sep 2002. 1, 2, 7, 10

[7] K. Keutzer, S.Malik, R. Newton, J. Rabaey, and A. Sangiovanni-vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," 2000. 2

[8] P. Glaskowski, "Pentium 4 (partially) previewed," Aug 2000. 2

[9] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of Latency-Insensitive Design," *IEEE Transactions on Computer Aided Design of Intregrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep 2001. 3, 4, 7, 10, 11, 13, 14, 21, 22, 24, 25, 26, 29, 33, 34, 36, 39, 41, 42, 43, 44, 47, 48, 49, 56, 57, 58, 91, 92, 96, 109, 110, 124, 132

[10] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *International Conference on Computer Aided Design*, 1999, pp. 309–315. 10

[11] L. Carloni and A. Sangiovanni-Vincentelli, "Performance Analysis and Optimization of Latency-Insensitive Systems," in *Design Automation Conference*, June 2000, pp. 361–367. 10, 11

[12] S. Sonalkar, C. Collins, and L. Carloni, "Design, Implementation and Validation of a New Class of Interface Circuites for Latency-Insensitive Design," in *Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2007, pp. 13–22. 10

[13] R. Lu and C. Koh, "Performance Analysis of Latency-Insensitive Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 3, pp. 469–483, Mar 2006. 11

[14] J. Millo, J. Boucaron, and R. Simone, "Another Glance at Relay Stations in Latency-Insensitive Design," *Electronic Notes in Theoritical Computer Science*, vol. 146, no. 2, pp. 41–59, 2006. 11

[15] T. Chelcea and S. Nowick, "Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols," in *Proc. Design Automation Conference*, June 2001, pp. 21–26. 11, 12, 57

[16] ——, "Robust Interfaces for Mixed-Timing Systems," *IEEE Trans. Very Large Scale Integration Systems*, vol. 12, pp. 857–873, Aug 2004. 11, 12, 13, 98, 105, 119

[17] M. Singh and M. Theobald, "Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures," in *Design, automation and test in Europe*, 2004. 11

[18] A. Edman, C. Svensson, and B. Mesgarzadeh, "Synchronous Latency-Insensitive Design for Multiple Clock Domain," in *IEEE International System-on-Chip Conference (SoCC)*, 2005, pp. 83–86. 11

[19] A. Chakraborty and M. Greenstreet, "Efficient Self-Timed Interfaces for Crossing Clock Domains," in *Proc. Intl. Symposium on Asynchronous Circuits and Systems*, May 2003, pp. 78–88. 11, 12, 13

[20] J. Mekie, S. Chakraborty, D. Sharma, G. Venkataramani, and P. Thiagarajan, "Interface design for rationally clocked GALS systems," in *ASYNC'06: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, 2006, p. 160. 11, 12, 13

[21] P. Teehan, M. Greenstreet, and G. Lemieux, "A Survey and Taxonomy of GALS Design Styles," *IEEE Design and Test of Computers*, vol. 24, no. 5, pp. 418–428, Jan 2007. 11

[22] D. M. Chapiro, "Globally-Asynchronous Locally-Synchronous Systems," *Ph.D. dissertation, Stanford University, Stanford, CA*, 1984. 12

[23] K. Y. Yun and R. P. Donohue, "A First Step Toward Heterogeneous systems," in *Proc. Int. Conf. Computer Design*, 1996. 12

[24] D. S. Bormann and P. Y. K. Cheung, "Asynchronous Wrapper for Heterogeneous Systems," in *Proc. Int. Conf. Computer Design*, 1997. 12

[25] A. E. Sjogren and C. J. Myers, "Interfacing Synchronous and Asynchronous Modules within a High-Speed Pipeline," *IEEE Trans. VLSI System*, Oct 2000. 12

[26] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, Dec 1998. 14, 25, 47

[27] A. Benveniste, "Compositional and uniform modeling of hybrid systems," *IEEE Transactions on Automatic Control*, vol. 43, no. 4, pp. 579–584, Apr 1998. 14

[28] C. A. R. Hoare, "Communicating Sequential Processes," in *Prentice-Hall International Series in Computer Science*, 1998. 141

[29] H. Andersen and M. Mendler, "A process algebra with multiple clocks," 1993. 141

[30] R. Cleaveland, G. Luttgen, and M. Mendler, "A algebraic theory of multiple clocks," in *International Conference on Concurrency Theory*, 1997, pp. 166–180. 141