Validating Families of Latency Insensitive Protocols

Syed Suhaib, Deepak Mathaikutty, and Sandeep Shukla
FERMAT Lab
Virginia Tech
Blacksburg, VA-24060, USA
Email: {ssuhaib,damathai,shukla}@vt.edu

David Berner
INRIA-IRISA
Campus Universitaire de Beaulieu
35042 Rennes, France
Email: david.berner@irisa.fr

Abstract— **With increasing clock frequencies, the signal delay on some interconnects in an SoC often exceeds the clock period which necessitates *latency insensitive protocols (LIPs)*. Correctness of a system composed of synchronous blocks communicating via LIPs is established by showing *latency equivalence* between a completely synchronous composition of the blocks, and the LIP based composition. Every time a new LIP is conceived, they need to be debugged and then proven correct. Mathematical theorems to establish correctness, though elegant, are error prone, and tedious to create for every new variant of LIPs. In this work, we present validation frameworks for families of LIPs, both for dynamic validation, useful for early debug cycles, and formal verification for formal proof of correctness. We believe this is a useful framework in the hands of designers trying to create new LIPs or optimizing existing ones for design convergence.**

I. INTRODUCTION

In the current and upcoming System-on-a-Chip (SoC) designs, intellectual property (IP) reuse is gaining increasing importance. Reusing pre-existing modules such as memories, processor cores, and dedicated hardware blocks chosen from an IP (Intellectual Property) library seems to be the only way to mitigate the problem of productivity crisis and shortening time-to-market cycles. Therefore, a significant part of the SoC design problem is in correct composition of these existing IP blocks [1]. However, what makes this more problematic is the fact that with the ever increasing clock frequencies, the synchrony assumption between IP blocks stops being valid. As the clock frequencies have crossed multi giga-hertz range, the clock period is too small for two communicating blocks to communicate signals across a long interconnect within such small clock period. This is because some interconnects are longer than the distance a signal propagates during a single clock cycle [2]. This problem has recently came into focus through a series of papers [2], [3], [4], [5], [6], [7], [8].

Although, a number of protocols, termed as Latency Insensitive Protocols (LIP)s have been published in the literature, no formal framework has been created to validate these protocols. The reason why one needs a framework where variants of these protocols can be quickly validated either through simulation or through model checking is as follows. These protocols are going through a continuing evolution phase. For example, [6] attempts to optimize and improve the protocols described in [2], [3], [4], [5] to obtain a more efficient and simpler protocol circuitry. In [7], a new protocol for pure Globally Asynchronous and Locally Synchronous (GALS) systems based on the earlier LIPs has been proposed. In [8], further simplification that obviates the use of specific protocol circuitry has been proposed. Although [5] offers a mathematical proof of correctness of their version of LIPs, when optimizations or extensions are made in the subsequent works, no such formal proof is usually offered. Due to the subtleties involved in the optimizations, it is plausible that the newly invented LIPs have serious flaws. We have experienced this in our attempts to optimize Carloni's protocol [3], [4]. As a result, we felt that there is a need for a framework where these protocols can be quickly modeled and validated either dynamically or statically through formal verification. This is the motivation for the current paper.

A. Solving the Long Interconnect Problem

Several approaches have been proposed to deal with the problem of long latencies in global physical chip interconnects in SoC design. One is a family of latency insensitive protocols (LIP) as in [5], where all modules are encapsulated with control logic blocks and possibly relay stations on the interconnects to make these interconnect delays transparent to the actual IPs. Another is to create packet based networks on a chip (NoC)s, also targeted at interconnect latencies [9].

However, designing the composition of IPs necessitate a refinement based design flow. In such a design flow, a synchronous model [10] of the system can be built where all interconnect latencies are assumed to be negligible. This follows from the synchrony hypothesis used in clock-synchronous hardware design, namely, communication latencies are negligible. From this synchronous model, necessary refinements are made to the design to render the design latency insensitive (LI). The correctness criteria is that the LI design be *latency equivalent* to the synchronous design. Informally, latency equivalence has to be first defined on signal pairs and then to process/system pairs. Two signals are said to be latency equivalent if the sequence of valid or *informative* events on the two signals are identical. In other words, if an observer observes the order of events on two signals, then discounting 'empty' events, the two signals will look the same. Since processes can be thought of as consumer and producer of events on signals, the same notion can be extended to systems, Two system models are latency equivalent if their outputs are latency equivalent given both are subjected to the same (or latency equivalent) input sequences. Latency equivalence is

formally defined in the preliminary definitions Section IV.

There are many ways one can ensure correctness of LIPs, and LI systems. Dynamic validation can be used to show that a system using LI techniques is latency equivalent to the completely synchronous model of the system which assumes zero delay communication. Although dynamic validation is appropriate for flushing out protocol design errors, such validation only covers for certain input sequences. Therefore, formal verification is a more desirable validation mechanism. In order to formally verify such protocols, the LI system as well as the synchronous idealization have to be modeled formally, and the latency equivalence has to be captured as a formal property. However, our experience is that model checking is very resource consuming [11]. Another way to confirm the correctness of such an implementation is to mathematically formalize it, as done in [5]. But mathematically proving the equivalence of two systems is a challenging task and not beyond mistakes. It requires complex mathematical proofs that are not straightforward to follow by others who want to confirm them, hence every new variation of LIPs cannot be validated easily using mathematical proof techniques. The best way is to provide designers with an easy to use framework to model and validate their protocols.

In this work, we propose a framework for validation of such systems. We target formal verification as well as simulation based techniques to verify the LI systems in our framework. For formal verification, we use the SPIN model checker to verify the correctness of an LI system, whereas for the simulation based technique, we use a functional programming based technique to validate the LI system. We contrast the two techniques and find that the SML based simulation for validation is a more convenient way to validate the framework, especially for debugging the early versions of the protocols.

Organization: This paper is organized as follows: In Section II, we discuss some related work. In Section III, the LI refinement methodology is illustrated followed by Section IV where we introduce the preliminary definitions and notations used in the paper. We also formalize the components of the LI protocol in this section. In Section V, we describe our framework along with its implementation in PROMELA and SML followed by conclusion in Section VI.

B. Main Contributions

The main contributions of the papers are as follows:
- Development of a framework for validating families of LIPs.
- Formal modeling and verification of a family of LIPs in SPIN.
- Modeling and simulation based validation of LIPs using a functional programming framework.

II. RELATED WORK

There are several approaches on how to create LI designs. Carloni et al proposed a "correct-by-construction" methodology to design latency insensitive systems for single clock

SoCs [3]. In their approach, all modules are encapsulated in a wrapper to form a "shell" that is latency equivalent to the actual process, without having to modify the internals of the original IP. This encapsulation is done by composing each process with an equalizer[1]. Relay stations are added along the long interconnections. They act like pipeline blocks to store and forward data, and contain at least two registers and control logic. Once the requirements of these relay stations are determined based on the long interconnect in the initial place and route, then placement and routing are done again with the relay stations. Several iterations for placement and routing may be needed in order to get a configuration that satisfies all interconnection constraints. In this paper, we refer to this approach as *relay-station based* approach.

In [8], we propose a different approach, which disposes of the relay-stations by adding some interface logic and wiring. While this approach requires more wires, it does not increase the number of components that have to be placed, and therefore uses less iterations for the placement and routing. We call this the *bridge-based* approach. Instead of placing relay stations along the long interconnects that connect two modules, we place a *bridge* process at the interface of the modules. A bridge is a composition of two processes: a *splitter* at the output of a "shell" and a *merger* at the input of the other "shell". The *splitter* and *merger* processes are formally defined in the preliminary definitions in Section IV.

All components of such LI designs are synchronous. The LI systems presented are targeting SoCs with a single master clock. Singh and Theobald generalize the LI theory for Globally Asynchronous and Locally Synchronous (GALS) systems [7]. In their approach all input and output signals are controlled by complex FSMs implemented in the wrapper. The communication network is implemented as an asynchronous system to connect modules with different clocks. Overall this approach is associated with heavy penalties in terms of implementation costs and performance.

Casu and Macchiarulo show how to reduce chip area compared to Carloni's approach [6]. They use a smart scheduling algorithm for the functional block activation and substitute relay stations with simple flip-flops. One disadvantage of this approach is that the schedule has to be computed a priori and depends on the computation in the process. If any change is made in any process, it may result in the change of the flow of tokens and may result in inconsistency with the current scheduling algorithm. In this case, the schedule has to be recalculated, which is expensive. We propose a validation framework for such LI protocols where they can be easily checked for correctness. We use two different techniques for validation: Formal verification using SPIN and SML based validation. This framework helps in validating such protocols that are continuously changing and evolving.

III. DESIGN FLOW TO LI REFINEMENT

In this section, we describe a transformation procedure to refine a synchronous system to an LI system shown in Figure 1

---

[1]The functionality of the equalizer is defined in the preliminary section
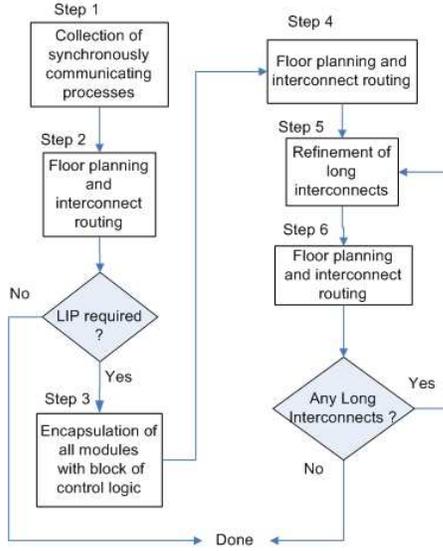
as a flow diagram.

Fig. 1.   Refinement steps to LI implementation

The steps to LIP refinement are as follows:
1. We start with a collection of synchronously communicating components. These components can be custom-made modules or IP cores.
2. Floor planning and interconnection routing are done to check for long interconnects. If all communication can be done in a single clock cycle, then there is no need for LIP refinement.
3. If long interconnects are present then all modules are encapsulated with a block of control logic. This encapsulation includes logic that controls the flow of the events, buffers, control stations, repeater stations etc to enable correct transmission of data.
4. Estimation using floor planning and interconnect routing is done again, this time with the encapsulated processes to relocate and evaluate the delays on the long interconnects.
5. After finding the delays on the long interconnects, the designer can then segment those long interconnects with additional processes containing buffers, latches, forwarding stations, etc to ensure that data is properly communicated through the long interconnects. Depending on the delay of the interconnect, the events can be compared from the point they are placed on the signal to the point they leave the signal.
6. Floor planning and interconnect routing is done again to ensure that no long interconnects exist in the system.

Once these synchronous components are composed together to form a latency insensitive system, our next goal is to check if the new LI system is functionally correct. Before we talk about the framework for validation for LI systems, we will first look at the background of the two validation techniques we use and present some preliminary definitions.

IV. BACKGROUND AND PRELIMINARY DEFINITIONS

In this section, we give the background for the SPIN model checker, functional programming as well as definitions we use throughout the paper.

A. SPIN background

SPIN [12] is a model checker used extensively for formal verification of systems. SPIN is used to trace logical design errors and to check the consistency of specifications. Like most model checkers, SPIN also verifies a system for all exhaustive paths. Its basic building blocks include asynchronous processes, message channels, synchronizing statements, and structured data. We use these basic blocks to write synchronous models. The communication is done through shared global variables. Since the processes run asynchronously in SPIN, we synchronize the execution of all processes in order to make our model behave synchronously. We illustrate and explain in detail the model of the clock controller at a later stage in the paper.

B. Functional Programming

In [13], we have presented a functional programming based framework for system modeling created on top of the Standard ML (SML) [14] language. Functional languages such as SML provide a clean and simple semantic model, which performs all computation by function application, thereby providing a more abstract notation to express computation. In our dynamic validation framework, we model the idealized synchronous model as well as its LI version and compare both the models by feeding input streams to both modules together and comparing their outputs for latency equivalence. The reason this framework is used will become obvious as we provide the definitions of various components of the LIPs in the subsequent sections. All the definitions can be readily recognized as recursive function definitions which can be directly mapped to SML.

C. Preliminary Definitions

In this section, we show some of the definitions we use in the rest of the paper. Let V be the set of data values and, T be a countable set of time stamps. Unless otherwise specified, in this paper, we assume $T = \mathbb{N} =$ set of natural numbers. An event $e \in V \times T$ is an occurrence of a data value with a particular time stamp. However, in the systems we consider, a special event called *absent event* denoted by τ may occur[2]. Therefore, the set of all events is denoted by E, where $\tau \in E$ and for all other $e \in E$, $e \in V \times T$. When $e \in V \times T$ it is called an *informative event*. A *signal* s is defined to be a sequence of events, often denoted as $e_1 e_2 e_3 \dots$ where $e_i \in E$.

For the preliminary definitions, if s is a signal, $s[i]$ denotes the ith event, hence either $s[i] \in V \times T$ or $s[i] = \tau$. The set of all signals is denoted by S. The signals can be either input signals or output signals of a process. We also distinguish *Stall signals* from all signals in the system. A stall signal s_t

---

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

is a sequence of boolean events, i.e., $s_t[i] \in Bool \times T$. The set of all stall signals is denoted by S_T. In our system, IPs are hardware modules that map input signals to output signals, therefore in this paper we refer to them as processes. A process p is a function $S^n \longrightarrow S^m$ where $n, m \in \mathbb{N}$. A synchronous system consists of these processes where communication and computation happens at the global clock. The communication among these processes is assumed to be zero-delay and each process takes one cycle for computation.

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

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

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

Definition 3: Given two tuples of n events and n signals respectively, $\oplus$ creates a tuple of n signals with an event appended to each signal.
$< e_1, \ldots, e_n > \oplus < s_1, \ldots, s_n > = < e_1 \oplus s_1, \ldots, e_n \oplus s_n >$

Definition 4: **Latency Equivalence**
The two signals s_1 and s_2 are said to be latency equivalent, $s_1 \equiv_e s_2 \Leftrightarrow \mathcal{F}(s_1) = \mathcal{F}(s_2)$, where
$\mathcal{F} : S \to S$ be defined as, $\mathcal{F}(s) = \sigma(s, 1, n)$ and,

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

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

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

Definition 6: **Feedback composition** [10]
Given a process $p: (S \times S) \to (S \times S)$ and $s_i, s_j, s_k \in S$, we define the feedback operator $FB_p(p)$ as:
$$FB_p(p)(s_i) = s_k \text{ where } p(s_i, s_j) = (s_j, s_k)$$
The signal s_j is an internally generated signal and the behavior of the feedback process is defined using fixed point semantics [10]. For simplicity, we define the feedback composition for a specific process with two input and output signals, though it can be easily generalized for processes with multiple inputs and outputs.

Definition 7: A vectorization function $\Upsilon_{i=1}^n(exp(i))$ is defined that evaluates the expression $exp(i)$ for i from 1 to n.
$$\Upsilon_{i=1}^n(exp(i)) = < exp(1), exp(2), \cdots, exp(n) >$$
where, $exp(k)$ is a textual replacement of i by k in $exp(i)$.

In this work, we are targeting the *bridge-based* approach, where all synchronous modules are encapsulated with an *equalizer*. An equalizer ($\mathcal{E}$) is a process that given n input

signals and a stall signal, it produces n output signals and n stall signals. The functionality of the equalizer can be divided into three modes:

1. *Disable mode*: In this mode, the equalizer is stalled by the another process through an input stall signal. The equalizer sends absent events on all its output signals and enables all the output stall signals using function *InsertStl* (shown in Definition 8).
2. *Absent mode*: In this mode, the equalizer receives an absent event on one of its input signals and its input stall is disabled. The equalizer sends absent events on all its output signals and stalls only those processes from which it received an informative event using function *InsertAbt* (shown in Definition 8).
3. *Present mode*: The equalizer receives informative events on all its input signals and its input stall is disabled. It places these informative events on the output signals using function *InsertEvt* (shown in Definition 8).

Definition 8: **Equalizer**
Given $s_1, \ldots, s_n \in S$ and $s_t \in S_T$, the equalizer $\mathcal{E}: (S^n \times S_T) \longrightarrow (S^n \times S_T{}^n)$ is defined as:
$$\mathcal{E}(s_1, \ldots, s_n, s_t) = eval(s_1, \ldots, s_n, s_t, 1, \cdots, 1)$$
where,
$eval(s_1, \ldots, s_n, s_{t1} :: s_{t2}, i_1, i_2, \ldots, i_n) =$
  if $(s_{t1} = false)$ then
    if $(\exists_{j=1}^n (s_j[i_j]) = \tau)$ then $InsertAbt \oplus evalnextindex$
    else $InsertEvt \oplus evalnextevent$
  else $InsertStl \oplus evalnextstall$

$InsertAbt = < \tau, \tau, \ldots, \tau > \odot \Upsilon_{j=1}^n(exp_1(j))$
$InsertEvt = \Upsilon_{j=1}^n(s_j[i_j]) \odot < false, \ldots, false >$
$InsertStl = < \tau, \tau, \ldots, \tau > \odot < true, \ldots, true >$

$evalnextindex = eval(s_1, \cdots, s_n, s_t, \Upsilon_{j=1}^n(exp_2(j)))$
$evalnextevent = eval(s_1, \cdots, s_n, s_t, \Upsilon_{j=1}^n(i_j + 1))$
$evalnextstall = eval(s_1, \cdots, s_n, s_t, \Upsilon_{j=1}^n(exp_2(j)))$

  $exp_1(j)$: if $(s_j[i_j]) = \tau$ then $false$ else $true$
  $exp_2(j)$: if $(s_j[i_j]) = \tau$ then $i_j + 1$ else i_j

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

In the *bridge-based* approach, *splitter* and *merger* processes are placed for communication of data through the long interconnects. We compose these two processes to form a *bridge* process as shown in Figure 2. The *bridge* not only ensures correct flow of events from one process to another, but also ensures that the delay in between the events is minimized. Each *bridge* process has one input signal and one output signal.

The $splitter$ and the $merger$ process are connected by n interconnects where n is the delay on the long interconnect. Hence, the $splitter$ process has n output signals. This process contains simple placement logic for the placement of events on these n signals. The splitter is implemented at the output of a process, and it places events on the corresponding signals.
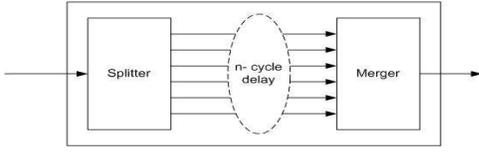
Fig. 2. Bridge

The splitter only places one input event on one of the output interconnects and absent events are placed on the rest of the signals at a particular time stamp. Assuming that there are i events on the input signal of the splitter, at every cycle, the ith event is placed on the nth signal based on a rotational scheme. For example, if the delay on the interconnect is 3 cycles, then in the current cycle, the first element will be placed on the first signal and absent events will be placed on the other two signals. In the next cycle, the second event will be placed on the second signal and absent events will be placed on the first and third signals and for the third event it will follow the scheme. After the third event is placed, in the following cycle, the fourth event will be placed on the first signal again. This rotation scheme will continue for the rest of the events. This functionality is illustrated by the formal definition shown below:

Definition 9: **Splitter**
Given $s \in S$, the Splitter $\mathcal{H} : S \rightarrow S^n$ is defined as:
$$\mathcal{H}(s) = spread(s, n, 1)$$
where,

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

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

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

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

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

Definition 10: **Merger**
Given $s_1, \cdots, s_n \in S$, the merger $\mathcal{M} : S^n \rightarrow S$ is defined as:
$$\mathcal{M}(s_1, \ldots, s_n) = ext(\ (s_1, \ldots, s_n),\ n,\ 1)$$
where,

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

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

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

V. Framework For Validation

It is essential to validate the functionality of the LI system that is formed by composing the components of the LIP with the synchronous system. We propose an easy to use framework for validating such LI systems. In this framework, we model the LI system along with its synchronous idealization and provide the same input signals to both the systems. These input signals can also be latency equivalent. We then model an $Eqcomparator$ process which is a reduced version of the Equalizer process. Similar to the equalizer, the $Eqcomparator$ process reads the informative events from the output signals of the two systems. The informative events on these signals are compared and checked to be latency equivalent. In the case when an absent event is seen on one of the output signals, it is discarded and the next event is considered on the same signal. The informative events on the two output signals are compared in sequence to ensure correct functionality.

Using this framework, we can take any system and its LI implementation and validate it for correct functionality by latency equivalence checking. This framework can be used for any proposed LI protocol to ensure the correctness for the system. For example, Casu and Macchiarulo's proposed approach for optimizing and using a scheduling algorithm to control the flow of tokens can be easily checked for correctness as the scheduling may change depending on the change of functionality of the system.

We have validated various examples of LI systems using our framework with two different approaches. We model a synchronous system and its LI implementation in PROMELA. Then we formally verify it by placing an assertion in the $Eqcomparator$ process. The assertion property states that the informative events from both systems are equal provided they are given the same input sequence. We show how we implement the component required for refinement to LI system in the following section. We also present another simulation based validation approach using this validation setup in the SML framework.

A. SPIN based description

In SPIN, the communication among the processes is implemented through global shared variables. A process may write to a variable and another may read from the same variable. Since, the SPIN model checker targets mainly asynchronous systems, and to model a synchronous system, we introduce a

clock controller process that controls the reading and writing of these variables for every clock. Hence, we divide the working of the processes into two phases, the read phase and the write phase. In the read phase, the processes read the values from its shared variables and in the write phase, new values are written on those variables. It is assumed that the communication is done in zero-time and all processes work concurrently as modeled. Temporary variables such as $done_i$ are used to denote the reading phase (when 0) or writing phase (when 1) of process P_i. Unless all the process complete reading or writing, depending on the phase, the clock controller will not change the phase.

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

Listing 1: PROMELA code for Equalizer

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

```
proctype Equalizer() {
    int valid_1, valid_2;
loop:
    Synchronize reading with other processes.
                    /* START OF READ PHASE */
    if
    □ (eventtype_A && valid_1 == 0) → valid_1++;
            Store value in temp1_A
    □ (eventtype_A && valid_1 == 1) → valid_1++;
            Store value in temp2_A
    □ (!eventtype_A && valid_1 == 1) → temp1_A = temp2_A
    □ else → skip
    fi;
                    /* Signal B can be written in a similar way */
    Synchronize writing with other processes.
                    /* START OF WRITE PHASE */
    if snipsTATE    □ (valid_1 > 0 && valid_2 > 0 && !stall) →
            valid_1--; valid_2--;
            Place events on output signals
    □ else → Place absent events on output signals ;
    fi;
    Set output stall signals based on valid_i values
  goto loop; }
```

The PROMELA code of the splitter process is shown in Listing 2. We assume that there is a two clock cycle delay on the interconnect where the splitter is placed. A temporary variable $place$ is defined that places the events from the input signals to either $signal0$ or $signal1$, depending on the current value of the $place$. These two signals connect the splitter and the merger. The $place$ variable keeps changing every clock cycle.

Listing 2: PROMELA code for Splitter

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

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

The PROMELA code of the merger process is shown in Listing 3. In this implementation, similar logic is used to read the values from the two incoming signals. The $place$ variable is offset in this module based on the delay on the interconnect. The values read from the signals are then placed on the output.

Listing 3: PROMELA code for Splitter

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

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

We formally verify the assertion for latency equivalence using the SPIN model checker for an example with two processes as shown in Figure 3. The example is of a simple

parity checker that checks if the input is a 1 or 0. The output of the system is based on the previous input's value.
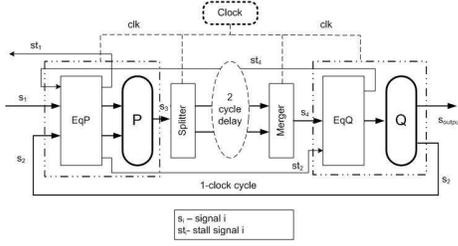

Fig. 3.   LI system with Bridge

An alternative to using formal verification, we use a simulation based technique to valide the LI systems in our proposed framework. We model this framework in SML and simulate for various test vectors.

B. SML based LIP description

In this section, we describe the components of the LI framework and its implementation in SML. A finite signal is modeled as generic list, whereas an infinite signal is written as delayed function application. In SML, for our convenience we formulate an event to be a list of two elements, where the first element is the value and the second element identifies whether the event is an informative event or an absent event (eg. $e_j = [3,1]$ is the jth event with 3 as the value and 1 as the identity of the event[3]). Hence, a signal can be formulated as a list of events. (eg: $s_i = [[1,1],[2,1],[3,0],\dots]$).

Following the earlier mentioned refinement methodology, we first encapsulate the synchronous components with an equalizer. The SML code of the equalizer is given in Listing 4. The equalizer reads one event from all the input signals of a process along with an event from the stall input. It then checks if all the events at a time are informative. The check for events is done through the $etypes$ and $info$ functions. The functionality setting the stall values for *Disable mode* is done by the $stallon$ function and the output is given by $e3$. The stall values when the equalizer is in *absent event mode* is set by $stallset$ function and the output is given by $e2$. Finally, the *valid mode* output is given by $e1$. The $equalizer$ process is then sequentially composed with the synchronous process to form the shell of the process.

Listing 4: SML code for Equalizer

```
fun equalizer() = fn s => fn st =>
                    f(s,st,indexstart(length(s))
fun f([],st1::st,_ ) = [] | f(_,[],_) = [] | f(_,_,[]) = [] |
     f(s,st1::st,i) =
let
     fun etype(x1::x2) = x2 | etype([]) = nil
     fun etypes[] = [] |
          etypes(x1::x) = etype(x1) @ etypes(x)
     fun info [] = false |
```

[3]1 corresponds to an informative event and 0 corresponds to an absent event

```
          info(x1::[]) = if (x1=1) then true else false |
          info(x1::x) = if (x1=1) then info(x) else false
     val allevents = e(s,i) /*Events from all signals*/
     val allinfo = if info(etypes(allevents)) = true
          then true else false
          /*True when all events are informative*/
     fun stalloff(0) = [] | stalloff(n) = [1] @ stalloff(n-1)
     fun stallon(0) = [] | stallon(n) = [0] @ stallon(n-1)
     fun flipval(x) = if x=1 then 0 else 1
     fun stallset([]) =[] |
          stallset(x1::x) = [flipval(x1)] @ stallset(x)
     val e1 = [allevents,[stalloff(length(allevents))]]
     val e2 = [tauevents(length(s)),[stallset(tags(allevents))]]
     val e3 = [tauevents(length(s)),[stallon(length(allevents))]]
 in
     (case(st1) of
        1 => (if allinfo = true
          then ([e1] @ f(s,st,incindex(i)))
          else ([e2] @ f(s,st,incempty(i,etypes(allevents)))))|
        0 => ([e3] @ f(s,st,i)) ―
       _ => [])
end
```

The next stage of the refinement methodology involves refining the long interconnects by inserting the bridge process. The delay on the bridge is modeled by the $Delayproc$ process that just delays the events by n cycles, where n is the delay on the interconnect. The SML implementation of the $splitter$ process is shown in Listing 5. An input signal and the interconnect delay is given to the $splitter$ process. One event is read from the input signal and $insertevent$ function places the event from the input signal to one of the interconnects and absent events are placed on rest of the interconnects. The events are placed in the rotational scheme as illustrated earlier.

Listing 5: SML code for Splitter

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

The SML representation of the merger is shown in Listing 6. The $extractevent$ function extracts one event from all signals at a time. Extraction of events from the signals is done in similar way as they are placed on the interconnects by the splitter.

Listing 6: SML code for Merger

```
fun merger(n) = fn s => g(s,n,1)
fun g([], n, i) = [] | g(x1::x, n, i) =
let
     fun extractevent([],n) = [] | extractevent(x1::x,n) =
          (case  (n) of
```

```
                1 => x1 |
                _ => extractevent(x, n-1))
    in
        if (i = n)
            then [extractevent(x1,i)] @ g(x, n, 1)
            else [extractevent(x1,i)] @ g(x, n, i+1)
    end
```

We compose all the components of the system after the refinement. The input sequence of the splitter and the output sequence of the merger are equivalent, since the order of events written by the splitter on the n output signals and the order of events read by the merger from its n input signals is the same. Therefore, the flow of events from the output of one shell across the long interconnect to the input of the corresponding shell is maintained. As the stall signals are dependent on the events received in the previous cycle from the processes to which these stall signals are connecting, they operate on a feedback semantics. We use the fixed point operator defined in the preliminary section to implement the feedback.

For this SML framework, we consider a larger case study of an adaptive modulator that consists of three IPs: regulator, convolutor and analyzer. The regulator module takes an input signal and a control signal and outputs based on the control signal by adding a threshold value. This output is then multiplied with a masking value by the convolutor module. The output of the system is given by the amplitude signal. The analyzer module outputs the control signal based on the input of the amplitude. Due to lack of space, we only show some code snippets in this paper. However, complete code listing can be downloaded from [15].

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

VI. Conclusion and Future Work

In this paper, we propose a framework for validation of LI systems. The LI systems along with their synchronous idealization can be modeled together and checked for the latency equivalence. We show two different techniques for validation using our framework. We model the entire framework in PROMELA and formally verify using the SPIN model checker. The latency equivalence is expressed as a formal property and verified for equivalence. We also show the validation using the functional programming based simulation technique where the framework can be modeled in SML and simulated for certain input vectors. The latency equivalence can be modeled by comparing the output of the two systems.

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

We also experimented with multiclock LI systems using our framework, but due to lack of space we do not show the details in this paper. A possible extension would be to modify the LI protocols to GALS system such that they could be easily validated in the framework.

References

[1] M. T. Bohr. Interconnect scaling - the real limiter to high performance ulsi. *IEEE Int. Electron Devices Meeting*, pages 241–244, 1995.

[2] L.P. Carloni and A.L. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro, Special Issue on Systems on Chip*, 22(5):12, October 2002.

[3] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *In Proc. International Conf. Computer Aided Verification*, pages 309–315, November 1999.

[4] L.P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Latency insensitive protocols. In *11th International Conference on Computer-Aided Verification*, volume 1633, pages 123–133, Trento, Italy, 07 1999. Springer Verlag.

[5] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. The Theory of Latency Insensitive Design. *IEEE Transactions on Computer Aided Design of Integrated Circuits and System*, 20(9):1059–1076, 2001.

[6] M. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Design Automation Conference*, 2004.

[7] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Design, Automation and Test in Europe (DATE'04)*, 2004.

[8] Syed Suhaib, David Berner, Deepak Mathaikutty, Jean-Pierre Talpin, and Sandeep Shukla. Presentation and formal verification of a family of protocols for latency insensitive design. Technical Report 2005-02, Virginia Tech, 2005.

[9] Luca Benini and Giovanni De Micheli. Networks on chip: A new paradigm for systems on chip design. In *Design Automation and Test in Europe*, 2002.

[10] Axel Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2001.

[11] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 2000.

[12] Gerard Holzmann. *The SPIN Model Checker*. Addison Wesley, 2004.

[13] D. A. Mathaikutty, Hiren D. Patel, and Sandeep K. Shukla. A functional programming framework of heterogeneous model of computation for system design. In *Forum of Design Languages (FDL 2004)*, 2004.

[14] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[15] LIP Fermat website. http://fermat.ece.vt.edu/LIP.html.