A Functional Programming Framework for Latency Insensitive Protocol Validation

Syed Suhaib, Deepak Mathaikutty and Sandeep Shukla

Virginia Polytechnic Institute and State University
Blacksburg, VA-24060, USA

David Berner and Jean-Pierre Talpin

INRIA-IRISA
Campus Universitaire de Beaulieu
35042 Rennes, France

Abstract

Latency insensitive protocols (LIPs) have been proposed as a viable means to connect synchronous IP blocks via long interconnects in a system-on-chip. The reason why one needs to implement LIPs on long interconnects stems from the fact that with increasing clock frequencies, the signal delay on some interconnects exceeds the clock period. 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. A design flow based on a synchronous composition specification, and stepwise refinement to LIP composition can be easily conceived, and a proof obligation to show latency equivalence between the synchronous specification and the refinement needs to be discharged. In this work, we propose a functional programming based framework for modeling and simulating LIP, and implement the semantics of various refinement steps in the programming model, so we can validate the LIP model against the original system within this functional programming framework. Such validation becomes easier due to the inherent denotational model of functional languages. We specifically use Standard ML to model the original system implementation as well as its latency insensitive version and compare the two by creating a model that contains both, giving them the same inputs and checking their outputs to be latency equivalent.

Key words: Latency insensitive protocols, functional programming framework, validation, SML.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

In todays embedded systems, clock speeds keep rising. The signal propagation speed however is not increasing, hence a growing number of designs hit a limit where some wires on the chip are as long as the distance a signal covers during one clock cycle. Since, the number of gates reachable in a single cycle does not change significantly, the percentage of the chip reachable within a single clock cycle is decreasing, and as a result we have reached a point where more gates fit on a chip than can be communicated in a single clock cycle [1]. In order to go past this limit, latency insensitive protocols (LIP) provide means to let components with a multiple clock cycle distance still communicate correctly [2].

In current SoC based design methodologies, reduced time-to-market dictates efficient reuse of complex components. This has led to the idea of developing libraries of Intellectual Property (IP) components. The integration of such complex IPs on SoC and communication between them has shifted the performance bottleneck of the system from computation within them to communication between them.

The impact of this shift of problem domains can be seen in state-of-the-art microprocessor designs. For example, the design of the hyperpipelined Netburst microarchitecture of Intel's Pentium 4 processor uses one of the first pipelines containing pipeline stages designed exclusively to handle wire delays. A *drive-stage* is dedicated to specifically handle the signal propagation without performing any computation [3]. Furthermore, recent studies using cache analysis tools predict that in a 35-nm design running at 10GHz, accessing a 4-Kbyte Level-1 cache requires about three clock cycles [4]. Increased relative interconnect length affects current memory-oriented microarchitectures that strongly rely on the low communication latency assumption [5]. Therefore, it is important to equip such models with protocols that make them latency insensitive (LI) and ensure proper functioning also for distances beyond one clock cycle.

There are many ways one can ensure correctness of LI systems. Dynamic validation can be used to show that a system using LI techniques is functionally equivalent to the completely synchronous model of the system which assumes zero delay communication. These tests are not reliable, since they only cover certain input sequences. Therefore, formal verification is more desirable validation mechanism. In order to formally verify such protocols, the LI system as well as its synchronous idealization have to be modeled formally, and latency equivalence has to be captured as a formal property. However, our experience is that model checking is very resource consuming, and verification of larger systems becomes difficult [6]. Also extension of the formal models is a tedious process. Another way to confirm the correctness of such an implementation is to mathematically formalize it, as done in [1]. 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 functional programming framework to validate such systems. A functional program is a function that receives the programs input as arguments and delivers the programs output as results. It has no internal state, which makes it free from any side effects. The validation framework for LI systems is easy to formulate and comprehend using functional programming, and can also be easily applied to large systems. We use Standard ML (SML) [7] to model the original system implementation as well as its LI version and check the output of both models to be latency equivalent. This SML based simulation for validation is a convenient way to validate the LI systems, especially for debugging the early versions of the protocols. In this work, we experiment with the *bridge-based approach* proposed by us in [6], which is a variation of the original LIP proposed by Carloni et. al.

Organization: The paper is organized as follows: In Section 2, we show the related work done using LIPs. In Section 3, we introduce the preliminary definitions and notations used in the paper, followed by the LIP refinement methodology illustrated in Section 4. In this section, we described the components needed for the LIP refinement and show how it is implemented in SML along with a case study. In Section 5, we show the LIP implementation for multiclock systems followed by the conclusion in Section 6.

2 Related Work

LIP for systems with long interconnection delays (i.e. greater than one clock cycle) were initially proposed by Carloni et al [8,1,9] for single-clock SoCs. In their approach, all processes are encapsulated in a wrapper to derive a process that is latency equivalent [1] to the actual process, without having to modify the internals of the original IP. 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 a control logic. The insertion of these relay stations increases the number of elements to route and requires additional space on the chip for their placement. Once it is determined where relay stations have to be added based on the length of the interconnects, placement and routing of the entire chip design now including the relay stations has to be redone. Several iterations for placement and routing are needed in order to get a configuration that satisfies all interconnection constraints.

All components of such LI designs are assumed to operate with the same clock. Singh and Theobald generalize the LI theory for Globally Asynchronous

[1] We define the notion of latency equivalence in the preliminary definitions section

and Locally Synchronous (GALS) systems [10]. In their approach, complex FSMs implemented in the wrapper control all input and output signals. 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 [2]. 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 a change of the flow of tokens. In this case, the schedule has to be recalculated, which is expensive.

In [6] we propose another modification of Carloni's approach, which involves removing the relay stations along the long interconnects and inserting extra wiring logic using a splitter and a merger process. This solution is generalized for multi-clock systems where communication is done based on a global clock and the process wrapper links the processes to the environment, irrespective of the local clock of the process. Since there are no relay stations, there is no requirement to place the relay stations along the wires, whereas the splitter and merger processes are placed on the interface of the process. We call this the *bridge based approach*. Here, we use this approach and show how to validate it in a functional programming framework.

3 Background and Preliminary Definitions

3.1 Functional Programming

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

We use Standard ML (SML) [7] to model the original system as well as its LI version and compare both the models. SML offers an excellent ratio of expressiveness to language complexity, and provides competitive efficiency. SML manages to combine safety, security, and robustness with a great deal of flexibility because of its type and module system. Other features of the SML based framework include:

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

and more reliable by automating memory management and garbage collection.

- SML does static type checking which detects many errors at evaluation time. Error detection is enhanced by the use of pattern matching and by the exception mechanism.
- The SML module system is an extension of the underlying polymorphic type system thereby providing separation of interface specification and implementation. These facilities are very effective in structuring large programs and defining generic, reusable software components.

3.2 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 $e_1e_2e_3\ldots$ where $e_i \in E$.

For the preliminary definitions, if s is a signal, s[i] denotes the i^{th} event, hence either s[i] $\in V \times T$ or s[i] $= \tau$. 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 *st* is a sequence of boolean events, i.e., $st[i] \in Bool \times T$ 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 are natural numbers. A synchronous system consists of these processes where zero-delay communication and zero-time computation among these processes happen at the global clock edge.

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

Definition 3.1 *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 3.2 *Given one tuple of m elements and another of n elements, $\bigodot$ creates a tuple of $m + n$ elements.*

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

Definition 3.3 *Given two tuples of n events and n signals respectively, $\bigoplus$ creates a tuple of n signals with an event appended to each signal.*

[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

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

Definition 3.4 ***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

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

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

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

Definition 3.5 ***Sequential composition***
Given two processes p_1: $S^u \rightarrow S^v$, p_2: $S^v \rightarrow S^w$ and $s_1, \cdots, s_u \in S$, we define the sequential operator $\circ$ as:

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

Definition 3.6 ***Feedback composition*** *[11]*
Given a process p: $(S \times S) \rightarrow (S \times S)$ and $s_i, s_j, s_k \in S$, we define the feedback operator $FB_p(p)$ as:

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

The signal s_j is an internally generated signal and the behavior of the feedback process is defined using fixed point semantics [11]. 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 3.7 *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 $exp(i)$.

4 LI Refinement Steps

In this section, we show a transformation procedure to design a LI system. The transformation of a synchronous system to a LI system is shown in Figure 1.

The steps to LIP refinement are as follows:

1. We start with a collection of synchronously communicating components.

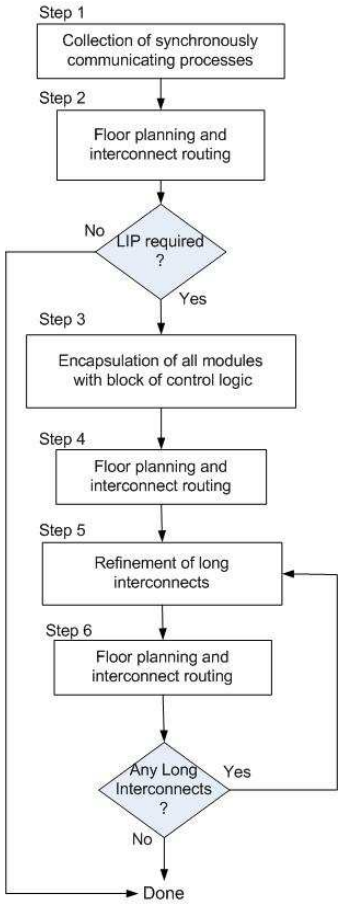

Fig. 1. Refinement steps to LI implementation

These components can be custom-made modules or IP cores.

2. Approximate floor planning and interconnection routing are done by design engineers to check for long interconnects. If all communication can be done in a clock cycle, then there is no need for LIP refinement.
3. All modules are encapsulated with a block of control logic. This encapsulation may include adding control logic that controls the flow of the events, buffers, control stations, repeater stations etc to enable correct transmission of data with the LIP. Once each process is encapsulated, verification is done to ensure its correctness, meaning that they behave similar to the original processes. By correctness here, we mean that if two processes are given the same set of input events, then the order of the informative events on its output signals are the same. We call this *latency equivalence*[3].
4. Estimation using floor planning and interconnect routing is done again, this time with the encapsulated processes to relocate and evaluate the

[3] Two signals are said to be latency equivalent if both signals have the same order of informative events

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.

4.1 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 as shown in Listing 1.

Listing 1: Finite and Infinite Signal

```
1 (* Definition of a finite signal *)
2 datatype signal = nil | 'a :: 'a list
3
4 (* Definition of an infinite signal*)
5 datatype infseq = nil | cons of 'a * (unit -> infseq)
```

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 j^{th} event with 3 as the value and 1 as the identity of the event[4]). Hence, a signal can be formulated as a list of events. (eg: s_i = [[1,1],[2,1],[3,0],...]).

The refinement steps for transforming a synchronous system to a LI system can be thought of as a two stage operation. The first stage involves encapsulating the synchronous components and the next stage involves refining the interconnects to make them consistent with the flow of events. These stages are shown in the steps in Figure 1. In the first stage, each module is encapsulated with an equalizer. An equalizer is a process instantiating template that given n input signals and a stall signal, it produces n output signals and n stall signals.

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

[4] 1 corresponds to an informative event and 0 corresponds to an absent event

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 4.1 ***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:*

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

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

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

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

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

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.

Listing 2 shows the implementation of the *equalizer* process in SML. The equalizer reads one event from all the input signals of a process along with an event from the stall input. It then checks if all the events at a time are informative. The check for events is done through the *etypes* and $info$ functions (lines 6-13). The functionality setting the stall values for *Disable mode* is done by the *stallon* function and the output is given by $e3$ (line 21). The stall values when the equalizer is in *absent event mode* is set by *stallset* function and the output is given by $e2$ (line 20). Finally, the *valid mode* output is given by $e1$ (line 19).

Listing 2: Equalizer

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

18
19  val e1 = [allevents,[stalloff(length(allevents))]]
20  val e2 = [tauevents(length(s)),[stallset(tags(allevents))]]
21  val e3 = [tauevents(length(s)), [stallon(length(allevents))]]
22 in
23 (case(st1) of
24  1 => (if allinfo = true
25    then ([e1] @ f(s,st,incrementindex(i)))
26    else ([e2] @ f(s,st,incrementempty(i,etypes(allevents))))
27       ) |
28  0 => ([e3] @ f(s,st,i)) |
29  _ => [])
30 end
```

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

The next stage of the refinement methodology involves refining the long interconnects by inserting processes that not only ensure correct flow of events from one process to another, but also ensure that the delay in between the events is minimized. The long interconnects are refined by inserting *bridge* processes (Listing 3). A bridge is formed by sequential composition of a *splitter* and a *merger* process (Figure 2). Each *bridge* process has one input signal and one output signal. The delay on the bridge is modeled by the *Delayproc* process (line 1) which delays the input by n cycles.

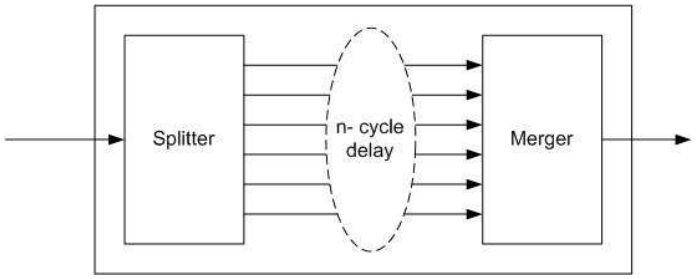

Fig. 2. Bridge

Listing 3: *Bridge* process

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

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

Definition 4.2 ***Splitter***
Given $s \in S$, the Splitter $\mathcal{H} : S \rightarrow S^n$ is defined as:

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

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

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

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

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

The SML implementation of the *splitter* process is shown in Listing 4. An input signal and the interconnect delay is given to the *splitter* process. One event is read from the input signal and *insertevent* function (line 6) places the event from the input signal to one of the interconnects and absent events are placed on rest of the interconnects. The events are placed in the rotational scheme as illustrated earlier.

Listing 4: Splitter

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

```
2
3  fun   f([],_,_) = [] |
4   f(x1::x, i, n) =
5  let
6   fun insertevent(_,j,0) = [] |
7       insertevent(y1,j,n) = (if n = j
8     then [y1] @ insertevent(y1,j,n-1)
9     else [[0,0]] @ insertevent(y1,j,n-1))
10 in
11  if (i = n)
12    then [insertevent(x1, i, n)] @ f(x, 1, n)
13    else [insertevent(x1, i, n)] @ f(x, i+1, n)
14 end
```

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

$\mathcal{M}(s_1, \ldots, s_n) = ext(\ (s_1, \ldots, s_n),\ n,\ 1)$ *where,*

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

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

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

The SML representation of the merger is shown in Listing 5. The *extractevent* function extracts one event from all signals at a time (line 3). Extraction of events from the signals is done in similar way as they are placed on the interconnects by the splitter.

Listing 5: Merger

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

```
13 end
```

After the refinement of all the components and the long interconnects of the synchronous system, all the components are composed together. The input sequence of the splitter and the output sequence of the merger are equivalent, since the order of events written by the splitter on the n output signals and the order of events read by the merger from its n input signals is the same. Therefore, the flow of events from the output of one shell across the long interconnect to the input of the corresponding shell is maintained. 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 feedback semantics. We use the feedback operator defined in the preliminary section to implement the feedback. Listing 6 shows the fixed point computation for the feedback semantics.

Listing 6: Feedback Process

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

4.2 Check for Correctness

Once we have the LI system and the original synchronous system, we have to verify if they are latency equivalent in order to satisfy the proof obligation. We do this by checking if the outputs of the two systems are latency equivalent given the same input sequence. The comparator is modeled which is a reduced version of the equalizer. This *Eqcomparator* process compares the order of informative events output by the two systems. 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. The LIP system satisfies the proof obligation if the output the two systems is latency equivalent, when given the same inputs. Figure 3 shows the setup of the problem. The SML code of the *Eqcomparator* is shown in Listing 7.

Listing 7: Eqcomparator

```
1  fun Eqcomparator() = fn s1 => fn s2 => compare(s1,s2,1,1);
2
3  fun compare([],_,_,_)= [] |
4   compare(_,[],_,_)= [] |
5   compare(s1,s2,i,j) =
6  let
7  val event1 = extractevent(s1,i);
8  val event2 = extractevent(s2,j);
9  val valid = if event1 = [] orelse event2 = [] then false else true;
10 val valpresent =
11  if valid = true
12    then if tag(event1) = [1] andalso tag(event2) = [1]
13      then true else false
```

```
14   else false;
15 fun comp(x,y) = if (x=y) then true else false;
16 in
17   if valid = true andalso valpresent = true
18      then (if comp(value(event1),value(event2)) = true
19        then [true] @ compare(s1,s2,i+1,j+1)
20        else [false])
21     else if valid = true
22        then if (tag(event1) <> [0] andalso tag(event2) <> [0])
23        then [false]
24     else if (tag(event1) = [0] andalso tag(event2) = [0])
25        then compare(s1,s2,i+1,j+1)
26     else if (tag(event1) = [0] andalso tag(event2) = [1])
27        then compare(s1,s2,i+1,j)
28     else if (tag(event1) = [1] andalso tag(event2) = [0])
29        then compare(s1,s2,i,j+1)
30     else [true]
31   else [true]
32 end
```

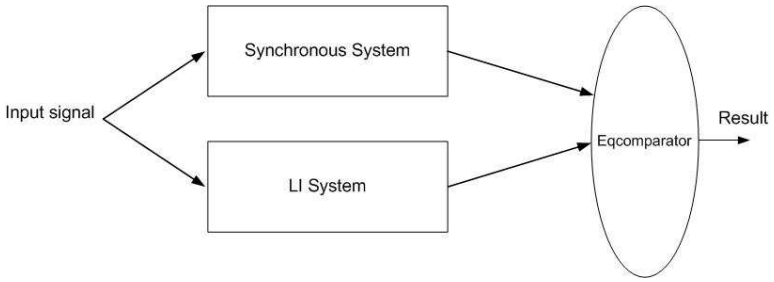

Fig. 3. Comparing synchronous system with its LI implementation

4.3 Case Study

We consider a case study of an adaptive modulator that consists of three IPs: regulator, convolutor and analyzer. The regulator module takes an input signal and a control signal and outputs based on the control signal by adding a threshold value. This output is then multiplied with a masking value by the convolutor module. The output of the system is given by the amplitude signal. The analyzer module outputs the control signal based on the input of the amplitude. The connections of these components are shown in Figure 4.

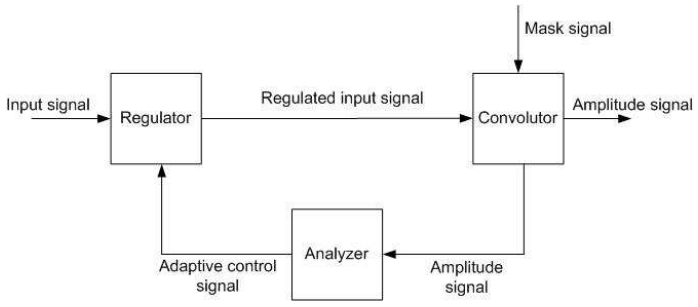

Fig. 4. Adaptive Modulator

With these three IPs, we follow the refinement steps described in Figure 1 to construct the LI system. We have these three synchronous components. Early floor planning and interconnect routing is done to find the delays on the interconnects. We assume in this case that the regulated input signal is a long interconnect. Hence, we encapsulate the modules with the equalizer and repeat floor planning to find the delay on the interconnects. The long interconnect is then refined by adding a bridge process. The new LI representation of the adaptive modulator is shown in Figure 5. The SML implementation is done using the components described in the previous section and is listed in the appendix. In order to check the correctness of the LI model, we create a model containing the LI implementation as well as the model with a zero communication assumption. We feed the same input sequence to both models and verify the latency equivalence of their outputs as described in Section 4.2.

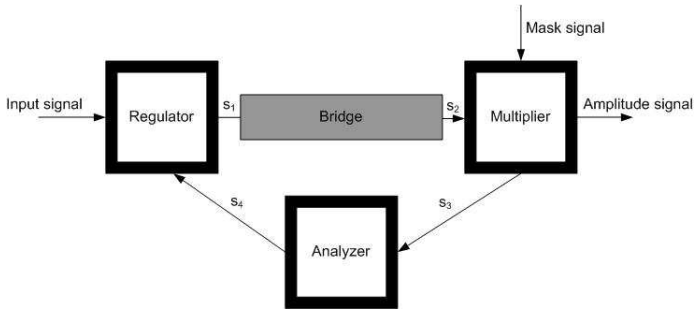

Fig. 5. LI based Adaptive Modulator

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

5 Multiclock extension to LIP

The LI systems proposed earlier have been mainly targeting single clock synchronous systems where all components operate on the same clock. We now consider extending the existing LI implementation for multiclock systems where different components with different clocks are connected via arbitrarily long interconnects. The need for a system with components having different clocks arises when different IP blocks from different vendor are integrated in the same system. At this time, however, we are only permitting the use of components with defined clock relations, also called rationally clocked systems. By clock relation, we mean that there is a known ratio of the evaluation

cycle[5] between different components. In the SML framework, the notion of clock is defined by the evaluation cycle of the processes. This approach therefore makes it possible to connect rationally clocked systems.

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

Definition 5.1 *Insert is a process, s.t.* $\mathcal{I}(s) = s'$ *where*

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

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

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

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

Definition 5.2 *Strip is a process, s.t.* $\mathcal{W}(s) = s'$ *where*

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

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

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

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

Listing 8: Multiclock Interface

```
1  fun Insert(n)= fn s1 => h(s1,n)
2  fun h([],_) = [] | h(x1::x, n) =
3  let
4   val sig1 = [x1] @ tausall(n)
5  in
6    [sig1] @ h(x,n)
7  end
8
9  fun Strip(n) = fn s1 => f(s1,n)
10 fun f([],_) = [] |
11   f(x1::x,n) =
12 let
```

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

```
13  fun dr [] = [] | dr(x::xf) = xf
14  fun drop ([],_) = [] | drop(s,1) = dr(s) |
15    drop (s,i) = drop(dr(s),i-1)
16 in
17  drop(x1,n) @ f(x,n)
18 end
```

6 Conclusion and Future Work

We propose a functional programming based framework using SML for the validation of LI systems against their original system implementations. The inherent denotational model of functional languages makes them well suited to formalize such complex protocols. In this framework, computation within the blocks can be changed without much additional effort whereas in formal verification any change of the model results in time consuming verification runs. We show a refinement methodology that defines how to transform a system consisting of synchronous blocks assuming zero delay communication to a corresponding LI system with long interconnects. An *Eqcomparator* comparator process is modeled that does a latency equivalence check between the outputs of the original system and its LI version given the same input sequence. We model a set of LI components, an *equalizer*, a *splitter* and a *merger* process, with which any deterministic synchronous system can be implemented. We extend this with processes *insert* and *strip* to multiclock systems where IPs with known clock ratios are also allowed.

Another possible extension for this is to allow unknown clock ratios. This would be another step towards being able to handle GALS systems where components can have clocks that are entirely unrelated.

References

[1] 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.

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

[3] P. Glaskowski. Pentium 4 (partially) previewed. *Microprocessor Report*, 14(8):10–13, 2000.

[4] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. *SIGARCH Comput. Archit. News*, 28(2):248–259, 2000.

[5] 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.

[6] 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, Virginia Tech, 2005.

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

[8] 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.

[9] 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.

[10] 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.

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

APPENDIX

Listing 9: SML Implementation for Adaptive Modulator

```
fun regulate([],_) = [] | regulate(_,[]) = [] |
    regulate(x1::x, y1::y) =
 if tag(y1) = [1] andalso value(y1) = 1
   then [[value(x1) - 10,1]] @ regulate(x,y)
   else [x1] @ regulate(x,y)

(* #1 and #2 corresponds to the first signal and second signal *)
fun Reg() = fn s1 => fn s2 => regulate(s1,s2)
fun Regulate() = fn s1 => fn s2 => fn st => Reg() (#1(Punzip()
 (Equalizer() s1 s2 st))) (#2(Punzip() (Equalizer() s1 s2 st)))

fun mul(_,[]) = [] | mul([],_) = [] |  mul(x::xf, y1::y) =
     if tag(x) = [1] andalso tag(y1) = [1]
 then ([[value(x) * value(y1),1]] @ mul(xf,y))
 else ([[0,0]] @ mul(xf,y));

fun Con() = fn s1 => fn s2 => mul(s1,s2);
fun Convolute() = fn s1 => fn s2 => fn st => Con() (#1(Punzip()
 (EqualizerTwo() s1 s2 st))) (#2(Punzip() (EqualizerTwo() s1 s2 st)))

fun comp(a,b) = if a > b then [1,1] else [0,1]
fun compsig([],_) = [] | compsig(x1::x,y1) = if tag(x1) = [1]
 then ([comp(value(x1),y1)] @ compsig(x,y1))
 else ([[0,0]] @ compsig(x,y1));
fun Alt(a) = fn s1 => compsig(s1,a)
fun Alternate(a) = fn s1 => fn st =>
 Alt(a)  (#1(Punzip2() (EqualizerOne() s1 st)))
```