A Process Algebraic View of Latency-Insensitive Systems

Hemangee K. Kapoor, *Member*, *IEEE*

Abstract—Latency-insensitive (LI) systems are those which can function correctly in spite of delays along its connecting wires. This delay is assumed to be a multiple of the clock period. The paper presents a single-clock process algebraic model for such systems. It gives the definitions for LI computational blocks and LI connectors. Important properties for these are shown to be satisfied. Composition of such modules can be done by the parallel composition operator of the process algebra. Conditions are given to check for liveness and deadlock freedom of LI systems. Comparison of latency equivalence between streams of events can be done using the model and this leads to a method of proving latency-equivalent modules. The paper is a step toward high-level specification and verification of such systems. The work can be extended to address more complex interconnections by modeling the underlying finite-state machines.

Index Terms—Latency-insensitive systems, process algebra, communicating sequential processes, trace equivalence.

1 Introduction

With the advance in semiconductor technology, we are able to pack more and more devices on a single chip. According to the Technology Roadmap for Semiconductors [1], we will be able to put more than 1 billion transistors on a single die. Also, the clock frequencies can be raised up to 19-20 GHz [2]. This is a good news to put more functionality on the same chip and still have smaller size devices. However, the threat comes from the long wires [3]. Wire delays dominate in deep-submicron (DSM) CMOS. These lengths can be managed locally within the module, but different Intellectual Property (IP) cores still need to be connected to each other. These interconnects are long and cause delays in signal transmission. Thus, the latency of this transfer increases. Traditional IP cores are designed to work with a fixed latency for input and output signals; however, the latency in an SOC is difficult to predict at design time and also may vary during place and route.

We therefore need IP cores which are insensitive to the latency on the input and output signals. In other words, no matter how long the signal takes to reach, the module must be able to receive it and compute correct results. This is similar to the delay-insensitive [4] assumption for asynchronous design [5]-[7], where the signal is assumed to take an arbitrary amount of time to travel. The difference with latency-insensitive (LI) design being that instead of arbitrary delays, the wires are assumed to have delays in multiples of the clock period [8] as they connect synchronous logic blocks.

• *H.K. Kapoor is with the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Guwahati 781 039, India. E-mail: hemangee@iitg.ernet.in.*

Manuscript received 20 Dec. 2007; revised 20 June 2008; accepted 25 Sept. 2008; published online 5 Dec. 2008.
Recommended for acceptance by S. Olariu.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-12-0645.
Digital Object Identifier 10.1109/TC.2008.214.

A synchronous specification is implemented using synthesis or custom design. After place and route, one must achieve timing closure. However, this becomes difficult as the wire delays dominate the design and a slight change in the floor planning changes the critical path estimate. Latency-insensitive designs therefore aim at solving this interconnect problem by inserting relay stations on the long wires. After place and route, one can perform timing analysis and update (increase/decrease) the number of relay stations required for a particular connection.

This paper tries to address the problem of latency-insensitive design by modeling the interconnect protocol in a process algebraic framework. The process calculus used is Communicating Sequential Processes (CSP) [9], [10]. We have used this existing framework and shown the feasibility of modeling LI modules. The individual modules being synchronous, we need to model time in our framework. Timed CSP [10], [11] is one option; however, as our requirement is not of continuous time, we have used the discrete timed version of CSP [10]. This helps to model time in terms of events occurring at regular intervals, modeled by the event $tock$.

We model the computational blocks and the connectors in this framework. The model assumes that all actions and computations are over before the next timed-event ($tock$) occurs. It thus models synchronous sequential logic, where the master clock governs the computations. The model is shown to satisfy many useful properties of an LI system. It also forms a basis for translation to circuits starting from the specification.

The paper gives a different view of LI systems and gives a process algebraic model of an already shown solution [12]. With the help of interactions defined in the model, one can visualize the kind of communication happening in LI systems and the tock event helps to demarcate sets of interactions.

The problem of delays and latencies exists in other fields (e.g., embedded systems) and the solution presented in the paper can form a basis to solve such problems. The model presented here is based on process interactions which are

closer to the real implementation, and hence easier to understand and analyze. Also, the model is flexible enough for future modifications and adaptations.

The remaining part of the paper is organized as follows. The next section discusses related work. We describe the language syntax in Section 3 and the model in Section 4. Analysis of the model is done in Section 5 where the properties satisfied by it are given followed by discussion in Section 6. Finally, we draw conclusions in Section 7.

2 Related Work

Latency-insensitive design is extensively studied in [2], [8]. In [8], synthesis method is given for synchronous systems that makes the design functionally equivalent to one which is tolerant to delays along long wires. This is done by developing shells around the IP cores and breaking the long wires into small segments separated by relay stations. A detailed theory behind this method appears in [12].

A framework for validating LI systems is given in [13]. Here, a given LI protocol is modeled in PROMELA language and verified using the SPIN tool. A similar approach to verify asynchronous handshake protocols connecting a DLX pipeline was done in [14]. In [13], simulation-based testing for a protocol is also demonstrated by modeling it in SML.

Formal modeling of LI systems using Marked graphs also appears in the literature. Marked graphs also known as Event graphs form a specific subclass of Petri Nets [15] where places have exactly one input transition and one output transition [16]. In [17], the relay-stations were modeled using SyncCharts [18] and model-checked. The Shell was modeled in Esterel and model-checked using Xeve [19]. Liveness and deadlock freedom were established as Petri-net marking conditions. Carloni [20] discusses on the throughput and performance of an LI system depending on the position and number of relay stations. A computational model of the LI system based on Marked graphs is described. It is discussed that the use of back pressure does not affect the maximum sustainable throughput of the system. On the other hand, a system using back pressure guarantees an implementation under any possible configuration of channel pipelining. Also, back-pressure mechanism replaces the need of using infinite length queues in providing correct implementation avoiding overflows.

A formal model of LI system using max-plus algebra [21] is given in [22] where a formally proved performance upper bound is shown to be achievable by an LI system. An alternative implementation is also provided that gives robust communication using back pressure.

A different approach to LI design is given in [23]. Here, the various functional blocks are activated using a scheduling algorithm. Each shell is stalled according to a periodic scheduling sequence which is stored in a local shift register. This reduces the routing resources and also area overheads of the sequential elements used for pipelining the long interconnects.

A generalized latency-insensitive design style is proposed in [24]. The work shows how to extend and generalize it to handle multiclock IP blocks. The two approaches suggested are: 1) to allow the synchronous modules to treat their input and output channels in a flexible manner and 2) to allow more flexible interconnections instead of point-to-point. In this regard, the work suggests an FSM-based approach for the wrapper design.

Designs of several low-latency mixed-timing FIFOs to interface systems are described in [25]. The system components may be either synchronous or asynchronous. FIFO designs are given to connect Sync-Sync, Sync-Async, and Async-Async modules. Also, the synchronous modules may be working at different frequencies. Other variants for latency-insensitive design appear in [26], [27] where the connectors are designed as elastic networks.

Other communication protocols to connect modules in the more general GALS framework also exist [28], [29]. Attempt to translate from weakly endochronous systems to delay-insensitive circuits can be found in [30].

Process algebras [9], [31] provide a well-studied framework for modeling and verifying concurrent systems. They all have facility to model time. Sometimes a single-clock timed process algebra is not sufficient to model systems. (GALS with modules having different frequencies is one example.) Multiclock process algebras are therefore developed to model such systems. Some examples are PMC [32], CSA [33], and CaSE [34] which are extensions of ATP [35] and CCS [31].

The scope of this paper is on latency-insensitive systems using a single clock, and hence, we use CSP having a single-clock event.

3 The Language

The syntax used is that of CSP [10]. The process alphabet is divided into input ($\mathcal{A}$) and output ($\mathcal{B}$). These inputs and outputs are called the input and output channels, respectively. Input of a value v on a channel c is denoted by $c?v$. We assume the values to be of a particular given data-type. Similar to input, an output is denoted by $c!v$ which outputs the value v over the channel c.

An example of a guarded choice (external/deterministic choice) is $[a \rightarrow P \,\Box\, b \rightarrow Q]$ where the process waits for either event a or for event b before behaving as P or Q. It can engage in only one alternative among a set of given choices.

Parallel composition of processes P and Q is denoted by $P \| Q$, where the processes synchronize on common actions. Concurrent execution of P and Q where no synchronization is required is described by the interleaving operator $P ||| Q$. Here, P and Q execute independently of each other and do not interact on any events apart from termination.

$skip$ is an event that happens instantaneously, $stop$ makes the process deadlocked, i.e., the process stops responding to inputs, and $error$ denotes a divergent state of the process.

Conditional actions are given by $P \triangleleft cond \triangleright Q$, where P is performed if the condition $cond$ holds, otherwise Q is executed.

We are modeling synchronous hardware processes, i.e., those using the concept of a clock. In Timed CSP [11], the associated time is a nonnegative real number. The time is thus dense or continuous. Synchronous hardware processes only require time (in terms of the clock signal) at intervals (discrete) and not as a continuous signal. We therefore use the discrete time version of Timed CSP that has the $tock$ signal which is a timed event occurring at fixed time intervals. Similar to the approach taken in [10], instead of

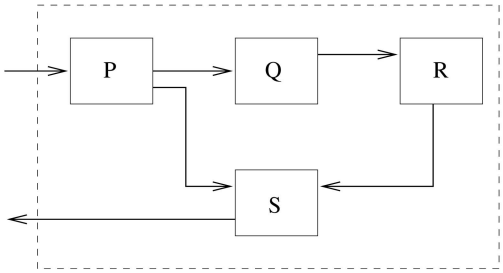

Fig. 1. Collection of modules in an LI system.

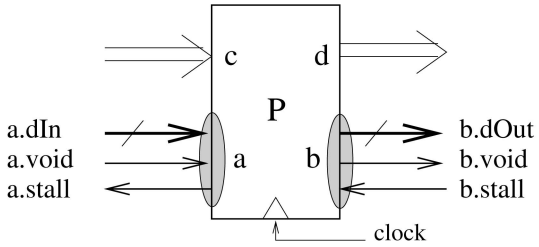

Fig. 2. LI module with input and output ports connected to channels.

adding time to the semantic model, we use fixed-interval time event as just another event in the untimed version of CSP. Because $\surd$ (tick) is used in CSP to denote successful termination, we use the symbol $tock$ to denote the passage of time.

The use of this $tock$ event is to denote passage of one unit of time. A process $P = tock \rightarrow a?v \rightarrow b!w \rightarrow stop$ waits for the tock event before accepting an input on channel a. It then outputs on channel b and then stops. A recursive process can be written as $P = tock \rightarrow a?v \rightarrow b!w \rightarrow P$. Here, after the initial tock event, the process does a sequence of one input and one output before engaging again in the tock event. In other words, between every input on a, a tock event must occur, i.e., a time passage of one unit has occurred.

To specify synchronous systems, we have an underlying assumption that all events between two successive tock events can be completed within the given time interval. In the above example, the input and output actions on a and b are completed before the next tock event is sent by the environment. In general, we assume that there is no combinatorial cycle (instantaneous loops). Every loop passes through a storage element which functions in lock step with the clock signal.

Under this timed specification, every process in a given system synchronizes with the tock event. The tock event is common to each component in the given composition of processes.

4 Modeling Latency-Insensitive Processes

A latency-insensitive (LI) system is a collection of modules (processes). For the scope of this paper, we assume that all the modules use the same clock. The modules communicate with each other over point-to-point connections. Each connection carries the data and control signals between the modules. Fig. 1 shows a sample LI system made up of four modules P, Q, R, S connected over a set of channels and communicating with its environment.

LI module is one which works correctly in spite of arbitrary delays on the connecting channels [8], [12]. The delay on the connectors is assumed to be a multiple of the clock period. For example, the signal may take three clock cycles to travel from P to Q and takes five clock cycles to reach module S. The modules/processes involved are expected to function in spite of these arbitrary delays.

Earlier work in LI systems has used trace like behavior to compare two LI processes. We take a similar approach in this paper, but in addition give a process algebraic description of such processes. This helps to derive the trace behavior and also understand the actual communications possible at a given time. The interactions can be understood by looking at the description and channel connections between the processes. The model also helps to check for liveness and deadlock freedom. Preliminary ideas on this approach appear in [36]. Marked graphs have also been used previously to establish liveness and deadlock freedom [16], [20].

The computational block described in the following subsections resembles the combination of the shell and the pearl (as in the basic LIP of [12]). In this paper, the pearl is a block performing computation and is represented as a function in the definition, and the shell is given as a set of interactions with the environment. Relay as given in [12] resembles the connector described in the following section.

4.1 LI Computational Block

In this subsection, we discuss an LI computational module. It is described by a process with input and output ports (Fig. 2). This process being synchronous waits for the clock tick ($tock$ signal in our language) 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;
3. the internal computation is longer than one clock cycle.

The latter reason need not be an issue, because at design time, the clock cycle is set to the critical path (i.e., computational path taking the maximum delay), and hence, the output is generated within one clock cycle. This falls inline with our assumption in the language that all computations between successive tocks are successfully completed (cf. Section 3).

Note that if the given process is unable to generate output at every tock, the receiving module may not get the required inputs at every tock.

The process is used to describe a synchronous module, and hence must read inputs after the tock arrives. There must therefore be a mechanism to inform the validity of inputs on the channels. We therefore divide each input port into three parts (Fig. 2):

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. We 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. Coming back to the 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.

4.1.1 Process Algebraic Model of the Computational Block

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, we use process which has state. This state will later 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).

A computational block is denoted by a process defined as follows:

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

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

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

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

$$
NXTSTL = P11 \triangleleft \forall i.xnw[i] \neq 0 \triangleright P10
$$

$$
REPEAT = P01 \triangleleft \forall i.xnw[i] \neq 0 \triangleright P00
$$

$$
\begin{aligned}
R01 &= P_{(x,xa,y,s,xnw,0,ynw)} \\
R11 &= P_{(x,xa,y,s,xnw,1,ynw)}
\end{aligned}
$$

$$
\begin{aligned}
P00 &= P_{(x,xa,y,s,xnw,0,0)} \\
P01 &= P_{(x',xa,y',s,xnw',0,1)} \\
P10 &= P_{(x,xa,y,s,xnw,1,0)} \\
P11 &= P_{(x',xa,y',s,xnw',1,1)}
\end{aligned}
$$

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).

We are now ready to explain the above definition of the computational block. The behavior is described by the following three steps: 1) clock tick, 2) input-output phase, and 3) repeat behavior.

1. Wait for the tock
2. (Input-output phase) The process checks the status of the variable st.
   {If st=1 : STLOUT}
   As there is a pending stall request, the process is not supposed to output data. It therefore sends a $void$! signal on all the output ports. This signal informs the corresponding receivers that the data on that channel are invalid (i.e., void). It also needs to stop the processes from sending new inputs as they will overwrite the unused inputs. This is done by sending a $stall$!1 to those input ports which have unused data. The other input ports can receive new data values (as there is buffer space), thus improving efficiency. In addition, the process reads inputs sent to it using subprocess IN. It then checks if it needs to stall for another cycle. If yes, it goes to the subprocess R11, otherwise to R01.

   Note that, this selective stall-out feature is new to our model. In earlier designs, the senders were stalled even if there was buffer space. Our design will thus be more efficient.
   {IN}
   The process reads either data or void values on all input ports and stall signals on each output port. While reading datum, if array x has unused datum, the new one is stored in xa. Note that synchronization on void channel occurs if its data channel does not have valid data. Similarly, stall receives "1" in case of stall. The values received are stored in the respective arrays for further use. If any stall requests are received during the input phase, then the process needs to store this information. The elements of the stall array are reset to "0" if no such requests arrive on the channel.
   {R11}
   It repeats the process behavior remembering that the next cycle is stall. The current cycle being a stall, the process does not compute new results (i.e., y array is unchanged). Hence, the variable ynw is kept as it is.
   {R01}
   It repeats the process behavior remembering that the next cycle is not a stall. The current cycle being a stall, it does not compute new results (i.e., y array is unchanged). Hence, the variable ynw is kept as it is.
   {If $\mathrm{st}=0$: DTOUT}
   There is no pending stall request. The process now checks if the output data are fresh or stale (using variable ynw).
   $\{\mathrm{ynw}=1\}$ If data are fresh (i.e., not yet sent over the output port), it sends the data.
   $\{\mathrm{ynw}=0$ This flag means that data were not computed during the last cycle due to unavailability of data on all input ports (i.e., some $xnw[i] = 0$). Hence, the process has to wait until all values are valid before continuing. It therefore sends a void signal on all output ports and stall signals on input ports that have valid values (this is again selective stall out that improves efficiency). The latter is required to prevent the unused data from getting overwritten. It then reads inputs using IN. If stall requests are received, perform NXTSTL, otherwise compute the results in REPEAT.
   {NXTSTL} If any output port has raised the stall signal, then the process has to stall itself. However, as this can happen in the next clock cycle, the information is communicated using the st state variable ($st = 1$): states P11 and P10. Whether new results are computed will depend on whether all inputs are available. Accordingly, one of P11 or P10 is selected.
   {REPEAT} If there are no stall request, then the st variable is set to "0" and new results are computed depending on the validity of all inputs.
   {P11}
   is the behavior where the process has to stall and the output data are not yet sent.
   {P01}
   is the behavior where the process does not stall and the output data are not yet sent.
   {P10}
   is the behavior where the process has to stall and the output data are already sent or is a stale value.

   New results are computed if all input values are valid, i.e., $(\forall i.xnw[i] \neq 0)$, in which case, new result y' is computed and stored in the process state. Otherwise, the results of the previous cycle are maintained (y). If certain input ports have received void data values, then the process cannot compute the results. In such cases, it maintains its current state (y).
3. The process is now ready for the next clock cycle (go back to step 1).

While composing different computational blocks, the events in DTOUT and IN synchronize with their corresponding signals on the receiver. These are instantaneous synchronizations (assumption that the complete input, output, and computation are completed within one clock cycle).

A detailed example showing the states and actions taken by the computational block is given in the Appendix. The example shows the execution flow over a series of clock cycles in Table 1.

4.2 LI Connectors

We have seen the LI protocol for the computational blocks. It guarantees correct functioning under absence of data over a sequence of clock cycles. The sender and receiver computational processes are thus taken care of.

The data, however, are traveling over wires (channels in our model). If the signal is able to reach the other module within one clock cycle (also modeled as instantaneous synchronization in our case), then no modifications are required.

However, 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 [8] 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

$$
\begin{aligned}
&C_{(x,xa,y,s,xnw,st,ynw)} \\
&\quad = tock \rightarrow \\
&\qquad ((STLOUT|||IN) \rightarrow \\
&\qquad\quad (R11 \triangleleft s = 1 \triangleright R01)) \\
&\qquad \triangleleft st = 1 \triangleright \\
&\qquad ((DTOUT|||IN) \rightarrow \\
&\qquad\quad (NXTSTL \triangleleft s = 1 \triangleright REPEAT))
\end{aligned}
$$

$$STLOUT = j.void! \rightarrow (i.stall!1 \triangleleft xnw \neq 0 \triangleright skip)$$

$$DTOUT = (j.dOut!y \triangleleft ynw = 1 \triangleright j.void!)$$

$$
\begin{aligned}
IN = [&i.void? \ \Box \ (i.dIn?x \triangleleft xnw = 0 \triangleright i.dIn?xa) \\
&\rightarrow xnw = xnw + 1] \\
&\rightarrow [j.stall?s \ \Box \ skip]
\end{aligned}
$$

$$NXTSTL = C11 \triangleleft xnw \neq 0 \triangleright C10$$
$$REPEAT = C01 \triangleleft xnw \neq 0 \triangleright C00$$

$$R11 = C_{(x,xa,y,s,xnw,1,ynw)}$$
$$R01 = C_{(x,xa,y,s,xnw,0,ynw)}$$

$$C00 = C_{(x,xa,y,s,xnw,0,0)}$$
$$C01 = C_{(x',xa,x,s,xnw',0,1)}$$
$$C10 = C_{(x,xa,y,s,xnw,1,0)}$$
$$C11 = C_{(x',xa,x,s,xnw',1,1)}$$

where

$$
\begin{aligned}
&\text{if } (xnw == 2) \\
&\text{then } \{x' = xa; xnw' = 1;\} \\
&\text{else } \{x' = x; xnw' = 0;\}.
\end{aligned}
$$

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).

5 Analysis

This section defines the behavior of LI module and states the properties enjoyed by such a system. The properties are stated only for the computational LI process, but they also hold for the LI connectors which are a special case of the generic LI process.

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 5.1 (Informative event). *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 5.2 (Void event). *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 ι event and vice versa.*

Definition 5.3 (Stall event). *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 5.4 (Reaction). *The set of actions which take place during one clock cycle is termed as a reaction. This is an interleaving of all possible input and output events. All these events are instantaneous during a particular clock cycle, and hence, the order between them is not important. We therefore denote the reaction by a set of events instead of an ordered sequence. The elements of a reaction include ι, or ϕ or σ.*

Definition 5.5 (Trace). *Trace is the sequence of reactions separated by the tock event. The set of all traces of a given process is denoted by $\mathcal{T}[\![P]\!]$.*

A general form for a trace is

$$\langle tock, r_1, tock, r_2, tock, \ldots, tock, r_n, \ldots \rangle,$$

where r_i is a reaction.

Example 1.1. For a more detailed example, consider a process with input ports a and c and output port b. One of its traces is

$$
\begin{aligned}
&\langle tock, \{\iota^a_{x_1}, \iota^c_{w_1}, \iota^b_{y_0}\}, tock, \{\iota^a_{x_2}, \iota^c_{w_2}, \iota^b_{y_1}\}, \\
&\quad tock, \{\iota^a_{x_3}, \phi^c, \iota^b_{y_2}, \sigma^b\}, tock, \{\phi^a, \iota^c_{w_3}, \phi^b, \sigma^b\}, \ldots \rangle,
\end{aligned}
$$

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. In this trace, the events inside the braces constitute a reaction.

Example 1.2. This example shows that void and stall events on certain channels lead to void and stall events on other channels in future clock cycles. This phenomenon is similar to the *procrastination effect* described in [12]. For simplicity, we will only consider example process with one input port a and one output port b. A sample trace is given below:

$$T_1 = \langle tock, \{\iota^a_{x_1}, \iota^b_{y_0}\}, tock, \{\iota^a_{x_2}, \iota^b_{y_1}\}, tock, \{\phi^a, \iota^b_{y_2}\}, \\ tock, \{\phi^a, \phi^b\}, tock, \{\iota^a_{x_3}, \phi^b\}, tock, \{\iota^a_{x_4}, \iota^b_{y_3}\}, \ldots\rangle.$$

Note that the void event in third clock cycle resulted in a void event in fourth cycle.

Because of this effect, it is difficult to use traces to compare processes. In the next subsection, we therefore concentrate on the trace along a single input (or output) channel to reason about latency equivalence.

As can be observed from Section 4.1, 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.

We will consider below certain other conditions under which we consider system deadlock.

First, we define what is meant by failure and divergence of an LI process.

Definition 5.6 (Failure). *After a given set of events, the process may be in a state in which it outputs void events on all its output channels. In other words, it is not able to send valid data values. The sequence of events that lead to such a state is considered failure of the process. The set of failures for process P is denoted by $\mathcal{F}(P)$.*

A process can come out from a failure state after it receives valid inputs on all input ports.

Example 1.3. Some failures for the example trace T_1 are

$$\langle tock, \{\iota^a_{x_1}, \iota^b_{y_0}\}, tock, \{\iota^a_{x_2}, \iota^b_{y_1}\}, tock, \{\phi^a, \iota^b_{y_2}\}, tock, \{\phi^a, \phi^b\}\rangle$$

and

$$\langle tock, \{\iota^a_{x_1}, \iota^b_{y_0}\}, tock, \{\iota^a_{x_2}, \iota^b_{y_1}\}, tock, \{\phi^a, \iota^b_{y_2}\}, \\ tock, \{\phi^a, \phi^b\}, tock, \{\iota^a_{x_3}, \phi^b\}\rangle.$$

Example 1.4. Consider a process with a feedback path apart from other inputs and outputs to the environment. It is necessary that the feedback path have some initial valid values, also called a token in a Petri net [37], to prevent the process from a failure state and a deadlock. This is because our model requires data on all input ports in order to compute output values. If the feedback path does not have initial token, then the process will wait for data on the feedback path before generating valid outputs. It is thus in a failure state.

The requirement of all inputs being valid can be relaxed with further modifications, cf. discussion on generic LI system in Section 6.3.

Definition 5.7 (Divergence). *If an LI process reaches a state in which it performs an infinite sequence of void input events and void output events, then it is said to have diverged.*

This is similar to infinite chatter in a CSP process, i.e., an infinite sequence of internal (τ) actions. The void input and output events in an LI system are similar to τ events because they do not carry valid data. The situation is a livelock, where no useful work gets done. One can however try to avoid livelock by assuming a fair environment, i.e., one which does not send stall and void signals indefinitely.

Example 1.5. Consider again a process with a feedback path apart from other inputs and outputs to the environment. It is necessary that the feedback path have some initial valid values to prevent the process from eventually diverging.

In absence of initial values on the feedback path, which is one of the process inputs, the process will not receive all the necessary inputs, and hence will not compute results (i.e., $ynw = 0$). This will lead to sending stall and void signals to senders and receivers (the process itself being one of them on account of feedback). It thus receives a void signal from itself which leads to further void signals, thus making it diverge.

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 5.8 (Stream). *A stream (S^T_a) is a sequence of events occurring in trace T, along a given channel (a) separated by the tock.*

Example 2.1. For example, the stream on channel a for trace T_1 in the above example is

$$S^{T_1}_a = \langle tock, \iota^a_{x_1}, tock, \iota^a_{x_2}, tock, \phi^a, tock, \phi^a, \\ tock, \iota^a_{x_3}, tock, \iota^a_{x_4}, \ldots\rangle.$$

We will drop the trace identity from the superscript, if it is implicit for the given situation, i.e., if the context is for trace T_1, we will simply write the stream as S_a.

Definition 5.9 (Informative stream). *Restricting the stream to informative events gives an informative stream. For a stream S, its informative stream is given by $S_I = S \uparrow \iota$.*

For the stream in Example 2.1, the informative stream is

$$S_{aI} = \langle \iota^a_{x_1}, \iota^a_{x_2}, \iota^a_{x_3}, \iota^a_{x_4}, \ldots\rangle.$$

Definition 5.10 (Latency equivalent streams). *Two streams are said to be latency equivalent (denoted $\equiv_L$) if they have the same informative streams. For streams S_1 and S_2*

$$(S_1 \equiv_L S_2) \text{ if } (S_{1I} \equiv S_{2I}).$$

Example 2.2. For example, consider the following streams S_1 and S_2:

$$S_1 = \langle tock, \iota^a_{x_1}, tock, \iota^a_{x_2}, tock, \phi^a_1, tock, \phi^a_1, tock, \iota^a_{x_3}, tock, \phi^a_1, tock\iota^a_{x_4}\rangle$$
$$S_2 = \langle tock, \iota^a_{x_1}, tock, \phi^a_1, tock, \iota^a_{x_2}, tock, \phi^a_1, tock, \phi^a_1, tock, \iota^a_{x_3}, tock, \iota^a_{x_4}\rangle.$$

Their respective informative streams are

$$S_{1I} = \langle \iota^a_{x_1}, \iota^a_{x_2}, \iota^a_{x_3}, \iota^a_{x_4}\rangle$$
$$S_{2I} = \langle \iota^a_{x_1}, \iota^a_{x_2}, \iota^a_{x_3}, \iota^a_{x_4}\rangle.$$

These streams are same, and hence, the original streams are latency equivalent: $S_1 \equiv_L S_2$.

Definition 5.11 (Latency equivalence). *Consider two latency-insensitive processes P and Q having the same alphabet. These two processes are latency-equivalent (denoted $P \equiv_L Q$) if the following conditions 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.*

More formally, if $\mathcal{A}$ is the input alphabet and $\mathcal{B}$ is the output alphabet of both P and Q, then

$$P \equiv_L Q \quad \text{if} \quad \forall T_p \in \mathcal{T}[\![P]\!], \quad \exists T_q \in \mathcal{T}[\![Q]\!] : ((\forall i : i \in \mathcal{A} : S_i^{T_p} \equiv_L S_i^{T_q}) \wedge (\forall j : j \in \mathcal{B} : S_j^{T_p} \equiv_L S_j^{T_q})) \text{ and vice versa,}$$

where $S_k^{T_p}$ ($S_k^{T_q}$) is the stream belonging to channel k of process P (Q) in trace T_p (T_q).

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 5.4 (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 (cf. Lemma 5.6). 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 in the definition, we can relax the second condition (of output streams being latency-equivalent), provided we know that the two processes are equivalent in terms of functionality. For example, consider two processes P and Q performing multiplication functions $mP()$ and $mQ()$, respectively. Given that $mP() \equiv mQ()$, then we need not compare the output streams for latency equivalence as P and Q are LI and we know that the information events on their outputs will be same.

5.3 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 give a better insight into the working of an LI process.

As all inputs are consumed at the same time, order of their arrival does no matter. Also, a second input must not arrive on a channel until the first is consumed.

Property 5.1. *Order of inputs among the ports does not matter.*

Inputs are read using synchronization on the respective channels. These are done using the interleaving operator, i.e., the order in which these synchronizations happen is not important. Thus, by definition, the order of arrival of inputs amongst the ports is not important. The computation can begin only after all inputs have arrived (and is not dependent on the arrival order). However, note that the order of inputs on a particular channel is important between two consecutive clock cycles.

Property 5.2. *Between two consecutive transitions on the same channel, there is a tock event.*

This follows from the above property. Synchronizations on channels occur once during each clock cycle. Before a second event can occur on the same channel, the processes must synchronize on the *tock* event, and hence the property. Note that an LI system is an implementation of a special case of a marked graph.

Property 5.3 (Safety). *Infinitely many actions do not happen between two tocks.*

This is a safety property to check that the processes do not diverge (i.e., perform infinite internal computations). It guarantees that there are finite number of events occurring between consecutive *tocks*. This property helps to satisfy the assumption of completing computation within one clock cycle. This relies on the hypothesis of absence of combinatorial loop, in the synchronous specification.

It can be verified by checking that the process restricted to only *tock* events refines the process which only performs *tock* events [11]

$$P \backslash (\Sigma \setminus \{tock\}) \sqsupseteq (TOCKS = tock \rightarrow TOCKS).$$

Next, we see the effects of stall events on the process behavior.

Property 5.4. *A stall event results in a void event.*

Proof. It suffices to prove that, for a given trace $T_1 = s \frown \langle r_n, tock, r_{(n+1)}\rangle \frown t$, $j \in \mathcal{B} \wedge j.stall?1 \in r_n \Rightarrow j.void! \in r_{(n+1)}$. Here, trace T_1 is prefix by some sequence s, followed by the events r_n, *tock* and $r_{(n+1)}$, which is then followed by sequence t.

$$\begin{aligned}
& j \in \mathcal{B} \wedge j.stall?1 \in r_n \\
\Rightarrow\ & \\
& s[j] = 1 \\
\Rightarrow\ & \{ \text{ execution of R11 or NXTSTL in clock cycle } n \ \} \\
& st = 1 \\
\Rightarrow\ & \{ \text{ execution of STLOUT in clock cycle } (n+1) \ \} \\
& j.void!
\end{aligned}$$

In other words, the process receives (during IN) a stall request in the current clock cycle, while it has already sent (during DTOUT) the output computed during the previous clock cycle. The stall information is saved (by NXTSTL) in the process state (st=1), and during the next cycle, the output of process is *void* (by STLOUT) instead of the data output. □

Property 5.5. *Stalls are acted upon one clock cycle later.*

This is a direct consequence of the above property. Stall received in nth cycle will result in stalling of outputs of $(n+1)$th cycle. There is thus a delay of at least one cycle before a process can stall itself.

Property 5.6. *Stalls ripple through a pipeline of computational blocks.*

When a process stalls, it will send stall signals to all its sender modules. These modules will then stall after one clock cycle (cf. above two properties).

When the process stalls, it will not consume data neither will it output computed results. In the case when data are not present on some inputs and present on certain others, these available inputs are not consumed until the remaining inputs arrive. In all such cases, we need to ensure that the input and output data arrays are not overwritten.

Lemma 5.1. *Data on output ports are not overwritten.*

Proof. Data to be output are stored in the state variable array y. Proof by induction on number of clock cycles.

Base case: clock cycle=1.

In the beginning, the array y is empty, and hence, during the first clock cycle, there is no danger of data overwriting. The new computed results can be safely stored in y.

Induction hypothesis: Assume that the lemma holds for n clock cycles, where $n > 1$.

Inductive step: clock cycle = $(n+1)$.

Here again, we have two cases:

1. Current cycle is not a stall: $\mathrm{st} = 0$. The array y has valid data if $ynw = 1$. The array y is used in subprocess DTOUT and modified by subprocesses P01 and P11. Data are not overwritten if it is used before it is modified. As can be seen from the definition of the LI process that P01 and P11 are invoked after the subprocess DTOUT, and hence data cannot be overwritten.
2. Current cycle is a stall: st=1. The process sends void and stall signals to other processes and neither uses nor modifies the array y, and hence, data cannot be overwritten. □

Lemma 5.2. *Data on input ports are not overwritten.*

Proof. Data that is input are stored in the state variable array x. The array x is used to compute the results in subprocess P01 and P11 during clock cycle n and is modified during the subprocess IN of clock cycle $n+1$.

Proof by induction on number of clock cycles.

Base case: clock cycle=1.

In the beginning, the array x is empty, and hence, during the first clock cycle, there is no danger of data overwriting. The new data values can be safely stored in x.

Induction hypothesis: Assume that the lemma holds for n clock cycles, where $n > 1$.

Inductive step: clock cycle = $(n+1)$.

The array x is used in the previous clock cycle during the subprocess P01 or P11. To avoid overwriting, we need to make sure that x was used to compute results. In other words, processes P01 and P11 were invoked in the previous cycle. This depends on whether or not the previous cycle was a stall. We therefore consider two cases:

1. Previous cycle was not a stall:
   Here again, we have two cases:
   a. All input values were received in the previous cycle, i.e., $\forall i.xnw[i] \neq 0$.
      In this case, the subprocesses using x were invoked and the array x was updated using the auxiliary array xa. If xa had valid data, it is copied to x and the xnw variable is set to 1, otherwise $xnw = 0$. During the current clock cycle, x or xa is updated depending on the value of xnw. Thus, if x still has unused data, the new data are stored in xa. The array x is thus not overwritten.
   b. All input values were not received in the previous cycle, i.e., $\exists i.xnw[i] = 0$.
      During the current cycle, data output has not taken place and stall signals are sent to processes that sent valid data in previous cycle. However, these senders will stall in the next cycle and have already sent data on the corresponding input ports. These data are stored in the auxiliary array xa, and hence, the array x is not overwritten. Eventually, when all data values arrive, the computation is performed and x, if required, is updated using xa.
2. Previous cycle was a stall.
   In this case, the subprocesses P01 and P11 were not invoked, and hence, the data in x are still unused. Data overwriting can be prevented in this case only if no inputs come on channel $dIn?x[i]$ during the process IN. In other words, during the current clock cycle while in subprocess IN, we must get inputs only on $i.void?$.

If the previous cycle was a stall, the process has sent stall events to all its senders. The current cycle will therefore be a stall cycle for the sender, who in turn will send void events to all its receivers (which includes our process under consideration).

We are thus guaranteed to receive void event, and hence, data will not be overwritten. □

Lemma 5.3. *Only one auxiliary array xa is sufficient.*

Proof. The scenario in which the auxiliary array xa is used is as follows: The process has not received values on all input ports, and hence must wait for the remaining values. In the mean time, it must ask the senders, which have sent valid values, to stall. However, as stalls are acted upon one clock cycle later, data items are on their

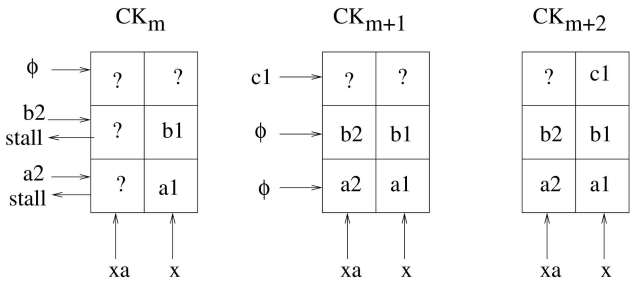

Fig. 3. Example showing use of auxiliary array.

way from these senders. We need to store these values in xa. However, the stalled senders will be continuously stalled until results are computed, and hence, no more values will be sent by them. Thus, a single array xa along with x is sufficient to store values. □

Fig. 3 shows an example process with three input ports and the signals on them over a period of three clock cycles. One can observe the use of auxiliary array on port c, and also that once stall is sent (on ports b and a), we receive void signals in the next cycle on these ports.

Lemma 5.4. *The process computes results after all the input values are valid.*

Proof. According to the computational process definition, the process computes new results during P01 and P11 (here, the ynw variable will be set to "1"). These are invoked during NXTSTL and REPEAT provided all inputs received on the ports are available and not yet consumed. This can be derived by checking the array xnw to indicate all valid values ($\forall i.xnw[i] \neq 0$). The xnw array is updated when each new input arrives and is updated when the results are computed (i.e., the input is consumed). Thus, the new results are computed when all input values are available.

If a certain input port, say k, does not have fresh data (i.e., $xnw[k] = 0$), then the process waits until it is available. This is done by stalling the other ports that have fresh values (in the subprocess DTOUT) and polling the port k (in the subprocess IN). These other ports are derived by checking the xnw array for nonzero entries (during DTOUT).

Note: The actions happening in the above scenario have a behavior similar to that of the *equalizer* described in [12]. Our model thus gives pointers to modules of the final implementation. □

Lemma 5.5. *Stale data are not used.*

Proof. New results are computed in subprocesses P01 and P11 when all inputs ports have new value (i.e., $\forall i.xnw[i] \neq 0$). The xnw array is updated during IN only after new data are read. Also, during IN, only if the array x is full, the values are copied in xa, thus maintaining x with fresh data values. The array x is used to compute the results, and hence, stale data are never used for computation. □

Lemma 5.6. *Process outputs the results of computation provided it has valid results and the receiving processes have not raised the stall signal.*

Proof. (Only valid results are output).

Valid results will be output if we have $ynw = 1$. This flag is set in P01 or P11 after computing results. If there is no new result, the process outputs voids on all the output ports (results are output only if there is no stall). Whether to send the output is decided beforehand, by using the variable st. The process does not output if $st = 1$. This variable, in turn, is set in the previous clock cycle if any of the receiving processes have raised the stall signal (cf. if $(\exists j.s[j] = 1)$ then NXTSTL else DTOUT). □

Property 5.7. *Every divergence is a failure and every failure is a trace.*

Follows from Definitions 5.5, 5.6, and 5.7.

5.4 Implementation

The previous subsections demonstrated various properties enjoyed by the CSP model of an LI process. We now discuss on mapping the model to a hardware implementation.

The main idea of latency-insensitive design is to connect synchronous blocks with each other over long wires. The blocks being synchronous hardware modules, the assumption of completing computation within one clock cycle is easily met.

A synchronous block computes results at each clock tick, and hence, to stall such a module, we need to do something extra. This can be done using a shell wrapper around the module [2]. The shell wrapper checks all the conditions and will appropriately stall the computation by stopping the clock. Stopping the clock is a feasible option [2].

We will now derive some of the components of the final implementation. Consider the definition of the computational block:

1. All the state variables used can be mapped to hardware registers. These are the latches holding the input and the output data. We also need latches to store the void and stall inputs.
2. The stall signal request can be implemented by stopping the clock of the computation block and adding logic to send void and stall outputs. This extra combinational logic goes inside the shell wrapper around the original computational block. The stall requests sent to the senders can be generated by combinational logic that checks for available buffer capacity. This logic (will improve efficiency and) will be different from that shown in [2].
3. New results are computed when all inputs are available. This can be checked using combinational logic which then enables the clock input to the computation block. In other words, allow the clock to run until a stall or void input is received. This makes the process *patient* as in [12].

5.5 LI System Properties

So far, we have defined and analyzed an LI process. We now discuss an LI system which is a collection of LI processes. This section discusses system-wide properties like liveness and deadlock freedom and helps to compare two LI processes. As the model is in CSP, we can use the tool FDR [38] (with some modifications) to check for equivalence and refinement.

Property 5.8 (Liveness-1). *Data received on all inputs will eventually be output.*

Proof. Follows from Lemma 5.2 and 5.6. □

Property 5.9 (Liveness-2). *If no stall requests are present, then the inputs will also not be stalled.*

Proof.

No stall requests
$\Rightarrow \quad \{\forall j.s[j] = 0\}$
subprocess R01 or REPEAT executed
$\Rightarrow \quad \{st = 0\}$
DTOUT executed (and not STLOUT).

□

Property 5.10 (Liveness-3). *Process does not send infinite number of stall events to its senders.*

Proof. The property is based on the hypothesis that the environment is fair, i.e., does not send void and stall signals indefinitely. From the definition of an LI process, say P, it can send a stall request to its senders under two circumstances:

1. When the receiver has requested a stall from P.
2. When input data are yet to arrive from a subset of senders on the input ports of P.

Assume that the process P is connected to a fair environment (which includes the sender and receiver processes of P).

In scenario 1 above, if the receiver is fair, it will eventually complete its computation and become ready to accept input data. Thus, it will remove the stall request from P, enabling P to send valid data. Thus, in scenario 1, P will not send stall events indefinitely.

In scenario 2, assuming the senders are fair, they will eventually send valid data on the inputs ports of P. After P receives valid data on all input ports, it will compute results and send data values instead of stall requests.

Thus, by construction, process P will not send infinite stall requests given that it operates in a fair environment, maintaining a system-wide liveness. □

Property 5.11 (Deadlock freedom). *The system deadlocks if all processes reach a failure state in the same clock cycle. To verify for deadlock freedom, we must make sure that at least one process is 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 (cf. Definition 5.6).

Also, in the case of cyclic interconnections in the module, at least one process must not be in failure state. This requirement is similar to that marked-graph model of the LI system, where the marked graph is live and deadlock free if for any cycle there exists at least one token. In other words, at least one process in the system can proceed.

Thus, by construction, the process will not be in a failure state as long as it receives the required number of valid data inputs, and thus will avoid deadlock. □

Property 5.12. *Composing two latency-insensitive processes, $P1 \| P2$, gives a latency-insensitive composition.*

Proof. Parallel composition of processes results in a composition which has some external input/output channels and some internal communication channels. We will consider two cases here:

1. External channels: Consider the case of input channels. Let a_i $(0 \leq i < n)$ be n external inputs to process P_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 P_2 receives void events on certain b_j channels
   $\Rightarrow$
   Process P_2 sends void and stall events to processes including P_1
   $\Rightarrow$
   On receiving stall request, P_1 sends void and (selective) stall events to external processes
   $\Rightarrow$
   Sender processes connected to channels a_i are stalled
   $\Rightarrow$
   The composition is thus latency $-$ insensitive on external input channels.

   A similar argument holds for external output channels.
2. 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. 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 latency-insensitive. □

Property 5.13. *Composing a latency-insensitive process with a latency-insensitive connector (storage element) gives a latency-insensitive combination. Here, process $P1$ is connected to C^m $(m \geq 0)$ which in turn is connected to process $P2$, where C^m is a composition of m connector blocks.*

This property follows from Property 5.12. The connector is an LI module and connecting any number of these will result in an LI composition. Continuing in this manner, we can connect any number of LI connectors to a computational block ($P1$) to get an LI composition. This will then be composed with the process $P2$; Fig. 4 shows the scenario.

6 Discussion

So far, we were concentrating on the CSP model and its properties. The exercise of doing this gives us insight into

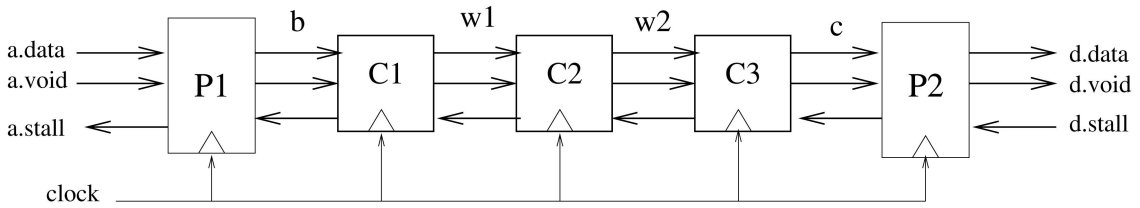

Fig. 4. Process P1 connected to process P2 over a long wire with three storage elements C1, C2, C3.

latency-insensitivity which helps to characterize LI behavior. This characteristic behavior can be used for building and verifying other models for LI systems.

The model of this paper requires input to be present on all input ports. To relax this requirement in order to give a generalized LI system [24], we need to do some modifications. All such issues are discussed below. In particular, we discuss about finding whether a process is LI, how to compare two LI systems, and how to extend the model for a generalized LI system.

6.1 Characteristic LI Process

The insight gained from the CSP model helps us identify as to what we mean by a latency-insensitive process. In other words, given a process description there is a way we can argue whether or not it is latency-insensitive. This can be handled by considering the interaction of the given process with a latency-insensitive environment. This analysis is feasible as the process is modeled in a process algebraic framework.

Consider a process P interacting with its environment E. The process has input and output alphabets $\mathcal{A}$ and $\mathcal{B}$, respectively, which form the output and input alphabets for the environment, thus forming a closed system (Fig. 5). Analysis using such interaction has been done earlier for delay-insensitive processes in [39].

We can analyze this closed system on the following points. The term process below means either P or E:

1. If a process wants to send data to the other, the receiver process must accept the data item.
2. The process cannot demand data but instead must wait (patiently) until data arrives.
3. After receiving necessary input data, the process cannot indefinitely refuse to output the results.
4. Data transfer must be acknowledged or negative-acknowledged. (In our model, negative acknowledgement is represented by the stall events.)
5. To ensure progress, failure of both must not happen at same time, i.e., $\mathcal{F}(P) \cap \mathcal{F}(E) = \emptyset$.

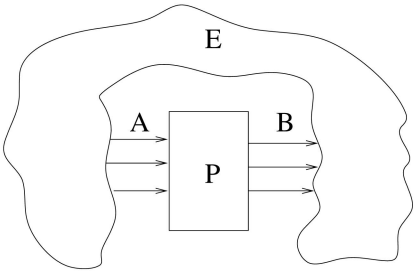

Fig. 5. Process and environment forming a closed system.

Testing an arbitrary process with an LI environment E can help to decide latency insensitivity for P. The environment E can be modeled with variable latencies by connecting an arbitrary number of connectors on its input or output channels. Processes passing this test will be, for example, those satisfying the model given in this paper, or processes with elastic FIFOs on its input and output channels.

6.2 Equivalence of LI Systems

An LI system consists of a set of LI processes. To compare any two given systems for latency equivalence, process algebraic manipulation comes of help. We can compute the resultant process using the parallel composition operator, and then, derive the traces using composition and hiding operators. The resultant traces of each system can then be used to compare the two systems for latency equivalence. This amounts to do a "weak bisimulation."

6.3 Generalized LI Systems

Throughout this paper, we considered processes with point-to-point links and assumed that data must be present on all inputs to generate data for all outputs. We can, however, modify the model thus making it more general. For such an extension to generalized LI systems [24], an FSM-based approach is needed. One can change process definitions of NXTSTL and REPEAT to incorporate such an FSM. The outputs will then be generated depending on the state machine, using only the necessary number of inputs (instead of all inputs) at each state of the FSM.

7 Conclusion

The paper has given a method to model latency-insensitive systems in a process algebraic framework. The arguments have been informal at some places; however, they are sufficient to have confidence in the model, and hence give a different method to specify such systems. It was shown that such modules can be described in the well-established process calculus CSP. This makes it possible to use standard tool such as FDR for equivalence and refinement checking.

We also discussed properties enjoyed by the model and argued about liveness and deadlock freedom using process traces and failures. The insight provided by the model helped to identify a characteristic LI process, which can be used to check for latency insensitivity of arbitrary processes.

The process algebraic model is easy to understand, and hence paves a way to translate from model to implementation. Synchronization of events during composition helps to understand the behavior of a larger composition. The storage elements along the long wire act like pipeline stages.

The paper assumed that the process needs to wait until all inputs are ready before it can compute results. More

TABLE 1
Flow of Execution for an LI Computational Process

Clock cycle	State (x,xa,y,s,xnw,st,ynw)	Actions	Remarks
1	x0, ?, y0, 0, 0, 0, 1	b.dOut!y0, a.dIn?x1, xnw=1 b.stall?0, y1=f(x1)	Input and Output
2	x1, ?, y1, 0, 0, 0, 1	b.dOut!y1, a.dIn?x2, xnw=1, b.stall?1, y2=f(x2)	Stall request on b
3	x2, ?, y2, 1, 0, 1, 1	b.void!, a.stall!1, a.dIn?x3, xnw=1, b.stall?0	Read new inputs Send void outputs Send stall to a
4	x3, ?, y2, 0, 1, 0, 1	b.dOut!y2, a.void?, b.stall?0, y3=f(x3)	Void input Send pending output
5	x3, ?, y3, 0, 0, 0, 1	b.dOut!y3, a.dIn?x4, xnw=1, b.stall?0, y4=f(x4)	Input and Output
6	x4, ?, y4, 0, 0, 0, 1	b.dOut!y4, a.void?, b.stall?0	Void input. Send output
7	x4, ?, y4, 0, 0, 0, 0	b.void!, a.void?, b.stall?0	Void output. Void input
8	x4, ?, y4, 0, 0, 0, 0	b.void!, a.dIn?x5, xnw=1, b.stall?0, y5=f(x5)	Void output. Read input
9	x5, ?, y5, 0, 0, 0, 1	b.dOut!y5, a.dIn?x6, xnw=1, b.stall?0, y6=f(x6)	Send output. Read input
10	x6, ?, y6, 0, 0, 0, 1	...	...

TABLE 2
Flow of Execution Showing Use of Auxiliary Array

Clock cycle	State (x, xa, y, s, xnw, st, ynw)	Actions
1	[x0,z0], [?,?], y0, [0,0], [0,0], 0, 1	b.dOut!y0, a.dIn?x1, c.dIn?z1, xnw=[1,1] b.stall?0, y1=f(x1,z1) Input and Output
2	[x1,z1], [?,?], y1, [0,0], [0,0], 0, 1	b.dOut!y1, a.dIn?x2, c.void?, xnw=[1,0], b.stall?0 Void input on c. Results not computed.
3	[x2,z1], [?,?], y1, [0,0], [1,0], 0, 0	b.void!, a.stall!1, a.dIn?x3, c.dIn?z2, xnw=[2,1], b.stall?0, y2=f(x2,z2) Input on a and c. Value on a stored in auxiliary array xa and later moved to x. Compute output. Stall a
4	[x3,z2], [x3,?], y2, [0,0], [1,0], 0, 1	b.dOut!y2, a.void?, c.dIn?z3, b.stall?0, y3=f(x3) Send output. Void input on a, as it was stalled
5	[x3,z3] [x3,?], y3, [0,0], [0,0], 0, 1	...

complex conditions on result computation (i.e., inputs available on only certain channels) can however be specified in the given framework. Also, the connector definitions can be modified to have fork and join connectors. This will help in specifying a generalized latency-insensitive system.

As part of future exploration, we can model multiclock LI systems, use storage queues of varying sizes, and also modify the protocol to make it general enough to accommodate variations, like 2ss and 1ss [40].

APPENDIX A

A.1 Example of a Latency-Insensitive Computational Process

Consider a process (P) with input channel a and output channel b. Let the computation performed by it be denoted by $y_i = f(x_i)$, where x_i is the input and y_i is the generated output. Initial values in input and output arrays are $x0$ and $y0$, respectively. Let the stream received on the input channel a be as follows:

$$S_a = \langle x1, x2, x3, \phi, x4, \phi, \phi, x5, x6, \ldots \rangle.$$

For simplicity, we have omitted writing the $tock$ event after every event in the above stream.

The module receiving inputs from P (over channel b) sends intermittent stall requests to P. The stall sequence received by P on the output port b is

$$\langle stall?0, stall?1, stall?0, stall?0, stall?0, stall?0, stall?0, stall?0, \ldots \rangle.$$

Table 1 gives the sequence of executions done over a period of nine clock cycles. The state column of the table gives the values of all the state variables at the current clock cycle. For example, during clock cycle 2, the state at the beginning of the clock cycle is (x1,?,y1,0,0,0,1) and at the end of the clock cycle is (x2,?,y2,1,0,1,1). Note that the latter state becomes the present state for the next clock cycle. The ? means the value is still to be initialized. In this example, the auxiliary array is not required as we have only one input port.

The output stream produced by P on the channel b can be derived from the table and is given below:

$$S_b = \langle y0, y1, \phi, y2, y3, y4, \phi, \phi, y5, \ldots \rangle.$$

Here again, for simplicity, we have omitted writing the *tock* event after every event in the above stream.

A.2 Example Showing Use of Auxiliary Array

Table 2 gives another example sequence showing use of auxiliary array xa. The process here has two input ports a and c and one output port b. The sender on c sends void inputs, and hence, the inputs on a need to be stored in auxiliary array.

ACKNOWLEDGMENT

The author is grateful to the anonymous reviewers for their comments.

REFERENCES

[1] A. Allan, D. Edenfeld, W.H. Joyner, A.B. Kahng, M. Rodgers, and Y. Zorian, "2001 Technology Roadmap for Semiconductors," *Computer,* vol. 35, no. 1, pp. 42-53, Jan. 2002.
[2] L.P. Carloni and A.L. Sangiovanni-Vincentelli, "Coping with Latency in Soc Design," *IEEE Micro,* pp. 24-35, Sept./Oct. 2002.
[3] J.A. Davis, "Interconnect Limits on Gigascale Integration (GSI) in the 21st Century," *Proc. IEEE,* vol. 89, no. 3, pp. 305-324, Mar. 2001.
[4] C.E. Molnar, T.P. Fang, and F.U. Rosenberger, "Synthesis of Delay Insensitive Modules," *Proc. Chapel Hill Conf. Very Large Scale Integration,* H. Fuchs, ed., pp. 67-86, Computer Science Press, 1985.
[5] A. Davis and S.M. Nowick, "An Introduction to Asynchronous Circuit Design," Technical Report UUCS-97-013, Dept. of Computer Science, Univ. of Utah, Sept. 1997.
[6] C.H.v. Berkel, M.B. Josephs, and S.M. Nowick, "Scanning the Technology: Applications of Asynchronous Circuits," *Proc. IEEE,* vol. 87, no. 2, pp. 223-233, Feb. 1999.
[7] S. Hauck, "Asynchronous Design Methodologies: An Overview," *Proc. IEEE,* vol. 83, no. 1, pp. 69-93, Jan. 1995.
[8] L.P. Carloni, K.L. McMillan, A. Saldanha, and A.L. Sangiovanni-Vincentelli, "A Methodology for Correct-by-Construction Latency Insensitive Design," *Proc. IEEE Int'l Conf. Computer-Aided Design (ICCAD),* pp. 309-315, 1999.
[9] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice-Hall Int'l Series in Computer Science, 1985.
[10] A.W. Roscoe, *The Theory and Practice of Concurrency.* Prentice-Hall Int'l Series in Computer Science, 1998.
[11] S. Schneider, *Concurrent and Real-Time Systems: The CSP Approach.* John Wiley and Sons, 2000.
[12] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli, "Theory of Latency Insensitive Design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 20, no. 9, pp. 1059-1076, Sept. 2001.
[13] S. Suhaib, D. Mathaikutty, S. Shukla, and D. Berner, "Validating Families of Latency Insensitive Protocols," *IEEE Trans. Computers,* vol. 55, no. 11, pp. 1391-1401, Nov. 2006.
[14] H.K. Kapoor, "Formal Modelling and Verification of an Asynchronous DLX Pipeline," *Proc. Fourth IEEE Int'l Conf. Software Eng. Formal Methods (SEFM),* pp. 118-127, 2006.
[15] C.A. Petri, "Kommunikation mit automaten," PhD dissertation, Faculty of Math. and Physics, Technische Universitat Darmstadt, Germany, 1962.
[16] F. Commoner, A.W. Holt, S. Even, and A. Pnueli, "Marked Directed Graph," *J. Computer System Sciences,* vol. 5, pp. 511-523, Oct. 1971.
[17] J. Boucaron, J.V. Millo, and R.D. Simone, "Another Glance at Relay Stations in Latency Insensitive Design," *Electronic Notes in Theoretical Computer Science,* vol. 146, no. 2, pp. 41-59, 2006.
[18] C. André, "Representation and Analysis of Reactive Behaviors: A Synchronous Approach," *Computational Eng. Systems Applications,* pp. 19-29, 1996.
[19] A. Bouali, "Xeve, an ESTEREL Verification Environment," *Proc. Int'l Conf. Computer Aided Verification (CAV),* pp. 500-504, 1998.
[20] L.P. Carloni, "The Role of Back-Pressure in Implementing Latency Insensitive Systems," *Electronic Notes in Theoretical Computer Science,* vol. 146, no. 2, pp. 61-80, 2006.
[21] F.L. Baccelli, G. Cohen, and G.J. Olsder, *Synchronization and Linearity: An Algebra for Discrete Event Systems.* Wiley, 1992.
[22] R. Lu and C.K. Koh, "Performance Analysis of Latency Insensitive Systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 3, pp. 469-483, Mar. 2006.
[23] M.R. Casu and L. Macchiarulo, "A New Approach to Latency Insensitive Design," *Proc. Design Automation Conf. (DAC),* pp. 576-581, June 2004.
[24] M. Singh and M. Theobald, "Generalised Latency Insensitive Systems for GALS Architectures," *Proc. Workshop Formal Methods for Globally Asynchronous Locally Synchronous (GALS) Architecture (FMGALS-03),* 2003.
[25] T. Chelcea and S.M. Nowick, "Robust Interfaces for Mixed-Timing Systems," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* vol. 12, no. 8, pp. 857-873, Aug. 2004.
[26] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of Synchronous Elastic Architectures," *Proc. Design Automation Conf. (DAC),* pp. 657-662, 2006.
[27] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary, "Synchronous Elastic Networks," *Proc. Design Automation Conf. (DAC),* 2006.
[28] D. Potop-Butucaru and B. Caillaud, "Concurrency in Synchronous Systems," *Formal Methods in System Design,* vol. 28, pp. 111-130, 2006.
[29] J.-P. Talpin and P.L. Guernic, "An Algebraic Theory for Behavioural Modeling and Protocol Synthesis in System Design," *Formal Methods in System Design,* vol. 28, pp. 131-151, 2006.
[30] S. Dasgupta, D. Potop-Butucaru, B. Caillaud, and A. Yakovlev, "Moving from Weakly Endochronous Systems to Delay Insensitive Circuits," *Electronic Notes in Theoretical Computer Science,* vol. 146, pp. 81-103, 2006.
[31] R. Milner, *Communication and Concurrency.* Prentice-Hall Int'l Series in Computer Science, 1989.
[32] H.R. Andersen and M. Mendler, "An Asynchornous Process Algebra with Multiple Clocks," *Proc. European Symp. Programming,* pp. 58-73, 1994.
[33] R. Cleaveland, G. Luttgen, and M. Mendler, "An Algebraic Theory of Multiple Clocks," *Proc. Int'l Conf. Concurrency Theory (CONCUR),* pp. 166-180, 1997.
[34] B. Norton, G. Luttgen, and M. Mendler, "A Compositional Semantic Theory for Synchronous Component-Based Design," *Proc. Int'l Conf. Concurrency Theory (CONCUR),* pp. 461-476, 2003.
[35] X. Nicollin and J. Sifakis, "The Algebra of Timed Processes, ATP: Theory and Application," *Information and Computation,* vol. 114, pp. 131-178, 1994.
[36] H.K. Kapoor, "Modelling Latency Insensitive Systems in CSP (Extended Abstract)," *Proc. Seventh Int'l Conf. Application of Concurrency to System Design (ACSD),* 2007.
[37] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. IEEE,* vol. 77, no. 4, pp. 541-580, Apr. 1989.
[38] F.S.E. Ltd, "FDR Tool," http://www.fsel.com/, 2009.
[39] M.B. Josephs and H.K. Kapoor, "Controllable Delay Insensitive Processes," *Fundamenta Informaticae,* vol. 78, no. 1, pp. 101-103, 2007.
[40] C. Li, R. Collins, S. Sonalkar, and L.P. Carloni, "Design, Implementation, and Validation of a New Class of Interface Circuits for Latency Insensitive Design," *Proc. Fifth ACM IEEE Int'l Conf. Formal Methods and Models for Codesign (MEMOCODE),* pp. 13-22, 2007.

Hemangee K. Kapoor received the bachelor of engineering degree in computer engineering from the College of Engineering, Pune, India, in 1998, the masters of technology degree in computer science and engineering from the Indian Institute of Technology Bombay in 2000, and the PhD degree in computer science from the London South Bank University, United Kingdom, in 2004. She is currently an assistant professor in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati. Her current research interests include process calculi, asynchronous systems, system-on-chip interconnects, and network-on-chip design. She is a member of the IEEE.