Another glance at Relay Stations in Latency-Insensitive Designs *

Julien Boucaron
INRIA Sophia-Antipolis

Jean-Vivien Millo
INRIA Sophia-Antipolis

Robert de Simone
INRIA Sophia-Antipolis

June 29, 2005

Abstract

We revisit the formal modeling of *relay stations*, which are specific connection elements used in the theory of Latency-Insensitive Design of Globally-Asynchronous/Locally-Synchronous systems. Relay stations are in charge of taking into account the physical mandatory latencies, while handling the regulation of signal/data traffic so as to avoid starvation, deadlock and congestion of local IP synchronous computation blocks. Since proposed by Carloni *et al*, the structure and behaviors of these relay stations have been amply characterized and analyzed. But previous works did not provide a fully formal and cycle-accurate description of these mechanisms, amenable to formal verification for instance (instead, mainly simulation models were developed). Due to the needed precision of the whole scheme we feel such a formal description might be needed. We describe such an attempt here.

1 Introduction

Long wire interconnect latencies may induce time-closure difficulties in modern SoC designs, with propagation of signals across the die in a single clock cycle becoming problematic. The theory of latency-insensitive design (LID), proposed originally by L. Carloni, K. McMillan and A. Sangiovanni-Vincentelli [21, 22], offers solutions for this issue. The theory can roughly be described as such: an initial fully synchronous reference specification is first desynchronized as an asynchronous network of synchronous block components (a GALS system). Then proper interconnect mechanisms are introduced to *re*synchronize the global system, but allowing specified (integer-time) latencies at interconnects, under the form of fixed-sized lines of so-called *relay stations*. These relay stations, together with *"shell"* wrappers around the synchronous *"pearl"* IP blocks, are in charge of managing the signal value flows. With their help proper regulation is performed between computation blocks that may be temporarily unable to run, either because of input data unavailability, or because of the inability of the rest of the network to store their results if they were produced. The second problem comes from the boundedness of hardware resources, and the fixed-size buffering capacity of the interconnects (the lines of relay stations).

Since their invention relay stations have been a subject of attention for a number of research groups. Extensive modeling, characterization and analysis were provided in [12, 15, 14]. Still, the modeling level has not completely reached a fully formal stage, so that proofs of correctness are still informal, either based on textual proof hints, or simulation model executions. We shall somehow use a paper by Casu et Macchiarulo [25], which provides such an (excellent) modeling, as our starting point. We depart from their description on a number of features, though (for instance they do not include the output functions as part of their FSM state machines describing the control structure of each relay station).

Each relay station can be conceived as a cell, to be part of a line of n, then composing the sectioned wire with a latency of n clock cycles. Relay stations implement a given protocol, that will in a sense be preserved by their chaining, only increasing the mandatory latency duration. Each station can receive a valid signal data from its predecessor (either a shell around an IP block or another station), and pass it down *in the next clock cycle* to its successor. The relay station can also receive in the reverse direction a regulation signal,

*This work is partially funded under a grant from Texas Instruments, Villeneuve-Loubet - FRANCE

implementing a *"back-pressure"* feature, to indicate that the successor node is unable to accept more data. In this case the station should refrain from sending its value and keep it instead. It should also still be able to receive the next one in this cycle (as the previous node was not warned of the congestion yet), and if necessary should propagate the back-pressure congestion signal to the previous node *in the next clock cycle*. The "next-cycle delays" are needed to respect the physical latency assumption. Of course there are also times where no valid data is transmitted from the previous node because upstream computations were temporarily halted due to lack of inputs. It should thus be noted that any relay station needs a capacity to hold *two* values simultaneously, in case it cannot propagate the current one while a new one simultaneously arrives. It can also be empty, if valid data are produced more slowly than consumed.

Currently the role of relay stations is two-fold: they implement the on-line scheduling scheme requested for proper handling of congestion risks, by back-pressure mechanisms; they also provide the temporary storage for data for as long as they cannot be forwarded further down the line. The second role is debatable: if the data were allowed to continue their route, they could be stored at the destination shell, if it would provide a dedicated buffer with the same size as the accumulated buffering capacity of all the relay stations on this line. Even better, moving all storage to a single spatial location would then ease the physical synthesis burden. This was noted in [25]. Of course the traffic regulation and the back-pressure mechanisms *should still be applied* in a mandatory fashion, since otherwise the end destination buffer could overflow. But they would only stall back data traffic and computation at shell level, not halfway through the interconnects. Back pressure mechanisms now show the net effect of retro-propagating information on the congestion and traffic jam reported *"downwards"*. They to do so only when needed, but *as early as feasible*, while respecting the latencies needed to travel through the long wires.

The paper is organized as follows:
In section 2 we recall briefly the basic contextual definition of synchronous circuits (for local components) and GALS systems (as networks of local synchronous computation components connected by unbounded buffers). We mention some initialization issues, solved as in [17] by the data valueless abstraction of GALS models into Petri Net Marked Graphs. It should be noted here that the body of theoretical results developed around Marked Graphs, also called Event Graphs in the literature, can provide a number of useful analytical results for the characterization of such systems [16, 5]. This is also true in the case of places with bounded capacity, and it provides answers to issues mentioned in previous papers on LID systems. In particular it provides sufficient conditions for proper initialization of data in lines, so as to guarantee liveness as absence of deadlock but also congestion altogether. On the other hand, Event Graphs (as all Petri Net subclasses) are inherently asynchronous as a concurrency model, and their application to scheduling and "maximal progress" remains for us to be investigated. Here again answers might already exist in the literature.
In section 3 we provide abstract requirements and formal constraints to be satisfied by relay stations models. We are starting from the model of [25], which itself somehow summarizes previous works. We provide our formal model, under the form of a SyncChart, with regular features and output signal clear timing specification. Our model is amenable to description in Esterel [7] or SyncCharts [3, 4], thereby allowing formal methods and model-checking techniques [9]. Of course this could also be possible by providing a direct netlist description in `blif` format for instance, but we gain syntactic flexibility, to describe easily the combination of several relay stations into a wire of great latency for instance.
In section 3.2 we specify formally a number of correctness properties, that can be established on a line of relay stations. Of course brute-force model-checking does not allow to reason on parametric models (where here the parameter would be the latency length n of the line), so we need to instantiate several constant length values.
We describe the shell wrappers (here very close to the version of [25]) in section 4. Again we model-checked them to establish correctness properties.
Related work in the direction of using static scheduling to optimize cycle allocation was started in [11, 24, 13], under the naming of *recycling* and inspired by software pipelining cycle allocation techniques. It extends and refers somehow to the paradigms of sequential circuit *retiming* [20].
We conclude with several open questions. The main topic for extension that attracts our attention is the following one: currently the design methodology starts from a monolithic synchronous specification. This is needed to retain several important synthesis techniques from commercial EDA flows. But if one can recognize that this seemingly synchronous description in fact contains informations indicating timing flex-

ibility and potential decomposition into smaller synchronous *"pearls"*, how could we effi ciently extend the approach to use this extra knowledge ? Here we are referring to so-to-speak *asynchronous* processes (with the word "asynchronous" here applied to the *computation model*), rather than to buffered connections (where the word "asynchronous" is applied to the *communication model*. Examples of such extra information could be provided by the user (as multirate/multiclock modeling extensions, or exclusive control modes) [1, 8, 27]. It could also be extracted by dynamic semantic analysis, as is done in the *iso/endochrony* theory of Benveniste *et al* [23, 6] (to the best of our understanding).

2 Preliminaries

Synchronous circuit: A synchronous circuit is associated with a clock. It has a signal interface consisting of three sets of (Boolean) input, register and output signals, and an internal state consisting of a set of (Boolean) registers (or flip-flops). On each clock *tick*, it produces current outputs and next-instant register values from the current values of inputs and registers.

Formally, a synchronous circuit is thus a structure $< \mathcal{I}, \mathcal{O}, \mathcal{R}, Out, Next >$, where

- $\mathcal{I}$ is a set of Boolean input variables $\{I_0, \ldots, I_{n-1}\}$. We call the vector $I =< I_0, \ldots, I_{n-1} > \in B^n$ an input event. It represents the valuation of all input variables at a given instant.
- $\mathcal{O}$ is the set of Boolean output variables $\{O_0, ..., O_{m-1}\}$, We call the vector $O =< O_0, \ldots, O_{m-1} > \in B^m$ an output event. It represents the valuation of all output variables at a given instant.
- $\mathcal{R}$ is the set of Boolean register variables $\{R_0, ..., R_{p-1}\}$, We call $R =< R_0, \ldots, R_{p-1} > \in B^p$ the current state. We also use the next-state $R' =< R'_0, \ldots, R'_{p-1} >$, using primed names.
- Out is a vector $< Out_0, \ldots, Out_{m-1} >$ of Boolean functions, $Out_j : (B^n \times B^p) \rightarrow B$. So each function Out_j defi nes the value of output variable O_j from the current values of input and register variables.
- $Next$ is a vector $< Next_0, \ldots, Next_{p-1} >$ of Boolean functions, $Next_j : (B^n \times B^p) \rightarrow B$. So each function $Next_j$ defi nes the next value of register variable R'_j from the current values of input and register variables.

Synchronous or asynchronous networks of synchronous circuits One can build larger circuits by setting local (IP) synchronous components in parallel, establishing desired point-to-point interconnections of inputs to outputs of different blocks. This is displayed in fi gure 1, if one assumes for the connections simple wires, and that all components run on the same clock. The result is then a compound netlist, homogeneous in nature with the local component synchronous circuits.

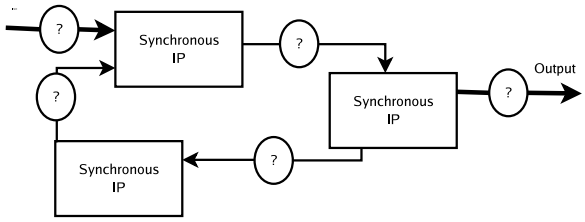


Figure 1: Network of synchronous IP blocks (synchronous or asynchronous)

On the other hand one can also assumes that local synchronous components are **not** globally synchronized, and that connections are established through *"ideal"* unbounded FIFO queues. This builds another interpretation of fi gure 1, as a global data-flow network. Now each component can be allowed to run only when all its input data values are present. The effect of its run is to consume one input value on each input

channel, and to produce one output value on its output channel. It can be conceived of as a fully unrestricted GALS system. We shall use this stage of representation only as an intermediate step for conceptual modeling.

As noted in [17], the unrestricted GALS model maps directly to Event/Marked Graphs (a well-known subclass of Petri Nets) when disregarding values carried as signal data. This association helps prove that, under some careful initialization conditions, this asynchronous version is functionally equivalent to the previous, fully synchronous one (see below the discussion on initialization).

Marked Graphs Also called Event graphs in the literature, they form a specific subclass of Petri Nets where places have exactly one input transition and one output transition [16]. In our case transitions represent local synchronous components, which indeed consume one data on each input channel, and produce one on each output channel in each step. With data abstracted as *"tokens"* the place marking represent the number of data currently contained in the interconnect FIFO queue.

Marked/Event Graphs are *"free-choice"* nets. Various executions only differ in relative schedulings of firings of individual transitions, and these behaviors are *confluent*: the firing of a given transition cannot disallow the one of another if it was previously allowed. Also, the sum of all places markings in a given graph cycle remains invariant all along any execution. A Petri Net (PN) is called *live* if any transition can still be executed (possibly after a number of steps) from any reachable marking. It had been proved in [16] that a Marked Graph is live if each graph cycle contains at least a token in one of its place. Figure 2 shows a Marked Graph associated with the previous GALS network (in its asynchronous form).

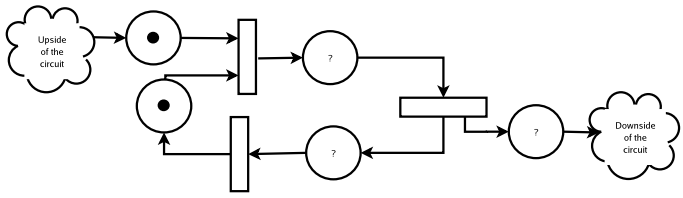


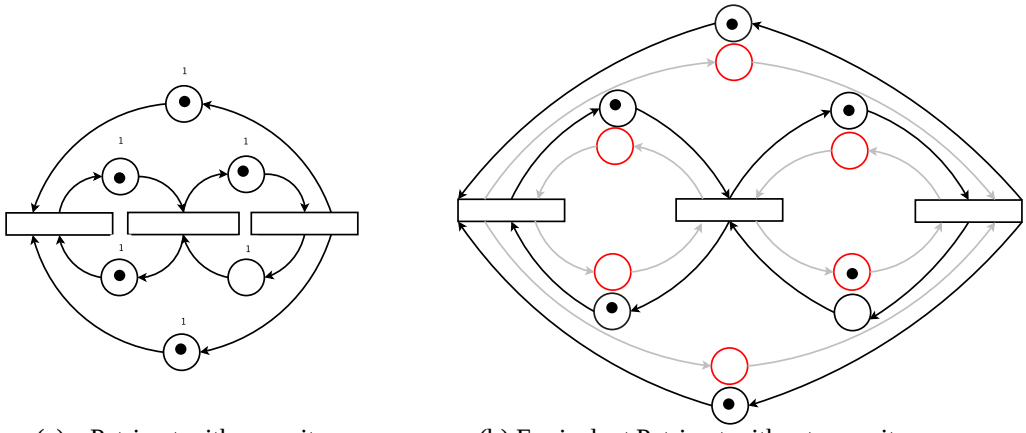
Figure 2: a live Marked Graph associated with the previous GALS picture

Marked Graphs with Place Capacities In GALS theories (such as Latency-Insensitive design and others), the purpose is usually to build a model *"in between"* the fully synchronous and asynchronous ones. In particular it is important in SoC design to be able to restrict interconnects so as to use only bounded space. The general philosophy is thus: first, desynchronize the fully synchronous specification; second, resynchronize it by careful scheduling mechanisms in a way that respect mandatory physical latencies, while using only bounded communication resources. At the abstract PN level, this boundedness can be modeled with place capacities (the scheduling issue will be dealt with elsewhere later).

Capacities are introduced in Petri Nets by requesting that a given place cannot hold more than n tokens, n being the capacity of that place. Capacities can be traced back to the foundation of PN history, without a clear seminal paper (see [5] for definitions and [2] for a proof of equivalence theorem). In fact it is immediate to replace a PN with capacity with another equivalent one *without* capacity by adding a new place for each existing one, with as marking as the difference between the original place capacity and its current initial marking. This new place is connected to transition in the reverse way as the original. Figure 3 displays a PN net with capacities (here of 1 for simplicity), and the equivalent PN with duplicated backward places.

Of course the bounded capacity raises new liveness problems, this time because of congestion and overflow instead of starvation and lack of available data tokens. Fortunately we can use the important fact that the above completion preserves the Marked Graph subclass, and inside this context solutions will be found. As will appear later, the latency-insensitive *relax-synchronized* version of our GALS system will possess a capacity of holding $2n$ data token on a connection line comprising of n relay-stations.

The final models produced in LID theory are (on first approximation) latency-bounded, resynchronized versions of marked graphs with capacities. In the sequel we shall call them *relaxed-synchronous* systems,



(a) a Petri net with capacity (b) Equivalent Petri net without capacity

Figure 3: From a Petri net with capacity (a) to an other Petri net without capacity but with the same behavior (b)

as they combine both synchronous features (all components and interconnects run on the same clock), and user-imposed interconnect minimal latencies (a constant integer delay for the line to transmit its signal/data values. While the data are still in transit, computation parts are paused by their surrounding shells, using clock-gating mechanisms. To respect the fixed-sized buffering ability a back-pressure congestion control protocol is applied across relay-stations.

Initial and well-formedness conditions We consider here various issues of proper initialization and structural well-formedness of the networks, ensuring for each semantics, be it synchronous or asynchronous, both starvation-freedom (or PN liveness), and congestion-freedom (or PN safety). We also briefly consider the quantitative issue of production rate (or throughput).

We recall the well-known fact that any graph can be decomposed as a directed acyclic graph (DAG) of strongly connected components (SCC), with a SCC possibly containing a single node.

Concerning **synchronous networks of synchronous components**, a valid signal/data must be present on each wire at the clock tick. In order to achieve this (while assuming it from the network primary inputs), one usually imposes that there is no combinatorial loop across the network. In other words *each loop in the network graph must cross a register*, which produces its output in the next clock cycle than it received it as input. Here the network graph consists of the local dependencies inside the components plus the interconnections between components. This is a strictly weaker condition as to impose that all component outputs are *latched* (as in Moore fashion), even though the second assumption is often recommended for composite design style, and is actually implicitly adopted in some of the GALS literature. Note here that the program of splitting up long combinatorial wires into sections is only fulfilled if not all local outputs are latched. Still, if it is the case one remains capable of turning unit delays into arbitrarily chosen delays.

Concerning **asynchronous networks of synchronous components** it is also the case that the network is live (so that all local components get fired infinitely often) iff *there is at least one token in each network cycle loop* (provided the primary inputs each provide an infinite stream of signal/data of course). This is a direct consequence of the result of Marked Graphs liveness. This matches closely the corresponding assumption on synchronous networks, provided the register is in fact a latch on a local output (but still not all outputs need to be latched, only one in each network communication cycle). The latched output can then be, in a sense, drawn from the local component to become the seed initial value of the interconnecting FIFO queue. Of course initialization with more values in the queues is feasible with liveness preserved (the more token the better in this case). But it is problematic to figure out how to obtain these seed values in general if starting from a fully synchronous specification with which to retain functional equivalence.

Considering **relaxed-synchronous versions**, where bounded capacity channels are replacing the unbounded buffering capacity of FIFO queues, a new kind of liveness problem is raised. Because of potential congestion, local computation blocks can now get blocked because their output channels are not ready to accept their results, which they could not store without overflow. This issue is theoretically solved by re-

questing that *the completed PN net do not allow any blank cycle*. Here the PN completion consists in adding the backward places to play the role of capacities. In other words each graph cycle in the completed graph should contain at least a token mark in one of its places. The net of figure 3 is a typical counterexample of this: with places each of capacity one, the net on the left is blocked; this is made explicit as blank cycles in the completed net on the right.

As we shall see later, a channel of n relay stations has a buffering capacity of $2n$ signal/data values. In the (frequent) case where the line is assumed to be initialized with only one value, then the virtual backward places all contain at least a token, thereby definitely disallowing blank cycles.

It has often been remarked in the GALS literature that, ultimately, a (simply connected) relaxed-synchronous network could run *no faster* than the speed of its slowest simple cycle loop. First, any SCC is restricted to the speed of its slowest cycle (after perhaps an initial phase where enough internal tokens can allow some parts to take "almost one lap in advance"). Then, whenever the part located upfront from a SCC starts running ahead, tokens accumulate at the entrance of this SCC until the bounded buffer gets filled, after which point there is no choice but to run the SCC behavior part. Similarly for the downstream parts, which needs data production from the upstream and SCC parts to be fed to run. It was established that the rate of the slowest loop was computed as the ratio of the number of data/token over the overall buffering ability over the loop.

3 Relay Station

We now come to the main part of this article. The purpose is to implement fixed-size communication channels that divide the long wires into sections, such that a signal/data can be propagated from one section to the next only in the next clock cycle. Similarly the signals needed to implement the congestion control back-pressure must also respect these traveling delays. To this end, *relay stations* were introduced in [21]. They are specific hardware elements that provide the proper interface between sections (and also the shells at the channel's ends). These elements must have some buffering activities, to store data *"on route"* of course, but also to park these additional data which might discover that because of congestion, the channel downstream cannot accept them.

3.1 Relay Station Modeling

Despite the number of publications describing relay stations in the literature, they are usually informally characterized. Neither their precise constraints representing the physical time requirements (in clock cycles), nor their formal model and their proper satisfaction is full described. The paper that comes nearest to this is [25]. However they do not use a pure synchronous modeling in their FSM (Finite State Machine). We shall deal now with all these issues.

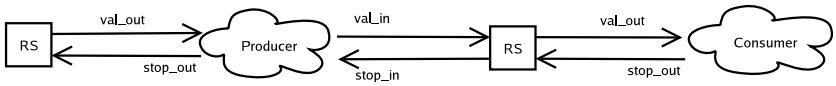


Figure 4: Relay Station - Block Diagram

We borrow from [25] the interface of input/output signals. It is depicted in figure 4. The data reception is represented by an input signal val_in being raised (it corresponds to $\neg\tau$ in the former articles on LID). It is a pure boolean signal (we can abstract the data values). Then the RS passes the data with a corresponding val_out signal. Concerning back-pressure, the RS can receive an halting order with the signal. The relay station receives input data with a valid signal $stop\_out$ being raised. It then transmits it with a $stop\_in$ signal (so $stop\_out$ is an input, and $stop\_in$ is an output).

Pseudo-physical requirement: It is important to note that signal/data cannot be propagated combinatorially from one section to the next:

- $val\_in \hookrightarrow_{next} val\_out$.

- $stop\_out \hookrightarrow_{next} stop\_in$.

On the other hand, there *can* be combinatorial relations between $stop\_in$ and val_in (resp. between $stop\_out$ and val_out, as they belong the same section.

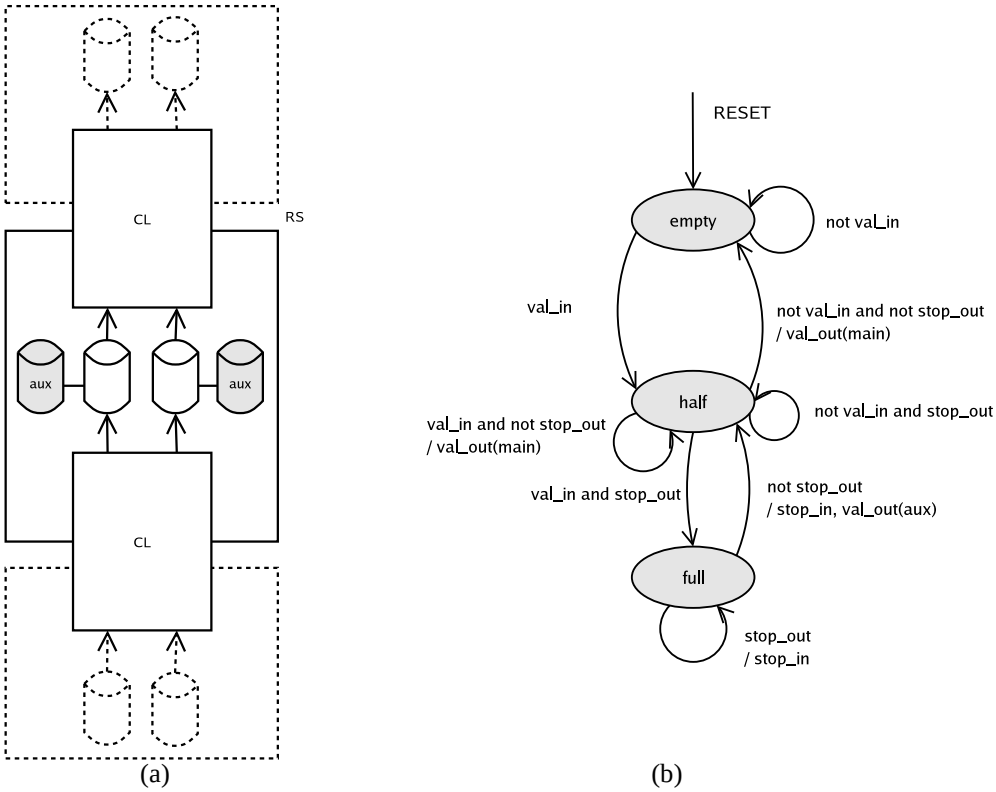


Figure 5: (a) Relay Station Structure (b) Relay Station SyncChart

So relay stations need registers (flip-flops for instance) to retain the signal between reception and propagation. In fact, as shown in [21], they need two such slots, in case a new data arrives while the current one cannot be propagated. Then, the congestion mechanism is supposed to guarantee that no further data can be received (and thus lost), because they are retained elsewhere upstream. This provides the abstract figure 5 (a).

3.1.1 Relay Station - SyncChart

We represent in figure 5 (b) the relay station as a SyncChart [3, 4], with explicit states, handling thus both the output and next state functions. Now we introduce this syncchart using for state encoding the number of registers free within the relay station.

The SSM (Safe State Machine) contains 3 states, corresponding to the occupation of the registers:

`empty` *when no data are currently buffered in the RS*; in this state the RS simply wait for a valid input data, and store it in its `main` register (goes to state `half`). $stop\_out$ signals are ignored, and not propagated upstream, as this cell can absorb traffic.

`half` *when it holds one data*; Then the RS cell only transmits its current, previously received signal data if ever it does not receive an halting $stop\_out$ signal (remember this combinatorial relation is correct, being inside a section). If halting, it retains its data, but must also accept a potential new one from upstream (as it has not sent any back-pressure holding signal yet). In the second case it becomes full, with the second value occupying its "emergency" `aux`iliary register. If the RS can transmit ($stop\_out$ false), it either goes back to `empty` or retrieve a new valid data in, remaining then in the same state.

On the other hand it still makes no provision to propagate back-pressure (in the next clock cycle), as it is still unnecessary due to its own buffering capacity.

`full` *when it contains two data*; then it raises in any case the $stop\_in$ signal, propagating to the upstream section the hold-out $stop\_out$ signal received in the previous clock cycle. If it does not itself receive a new $stop\_out$, then the line downstream was cleared enough so that it can transmit its data; otherwise it keeps it and remains halted.

NB: A signal is emitted (denoted by /) only when true, otherwise false.

Discussion With this precise cycle-accurate model, one can for instance wonder whether it would be feasible to improve the design to be able, while `full`, to both propagate its current data and accept a new one, remaining full. Of course this should be useless in practice, because the val_in signal could not be received (since the previous cell, when warned of its $stop\_out$, blocks its val_out to become the current RS's val_in). But if the RS is connected to another element, the shell for instance, the constraint $stop\_out \Rightarrow \neg val\_out$ has to be checked and guaranteed, or at least appropriate behavior must be checked. This can easily be done using trivial model-checking on our formal description.

3.2 Correctness properties and formal verification

Keeping with the kind of remarks of the previous discussion, one can phrase a number of correctness properties to hold on a relay station, or a line of relay stations (or later, a network comprising shells and pearls). Remember that correctness criteria for liveness (seen as freeness from both deadlock and congestion) were already established as PN graph markings conditions, linked to data initialization in section 2. Instances of additional properties are:

- relay stations cannot overflow or underflow.
- data order is preserved.
- at any point in time, the number of valid data produced from a line is bounded relative to the number entered:

$$\#(val\_in) + Init\_line \leq \#(val\_out) \leq \#(val\_in) + Init\_line + 2 \times length\_line$$

  where $Init\_line$ is the number of data initially residing in the line of RSs, and $length\_line$ is the number of RSs.
- a line of n relay station cannot notify congestion to its source unless it receives enough similar back-pressure signals, given its initial content.
- conversely, a line receiving enough back-pressure hold-out signals and data will eventually get fi lled and notice congestion.

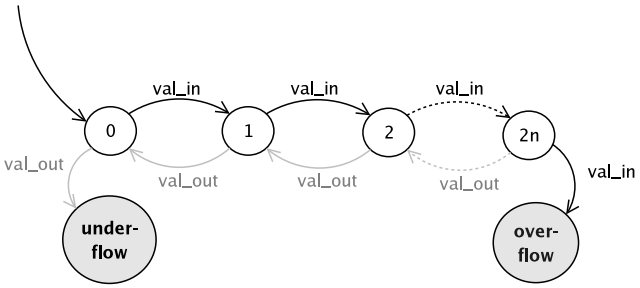


Figure 6: Overflow, underflow observer for RS

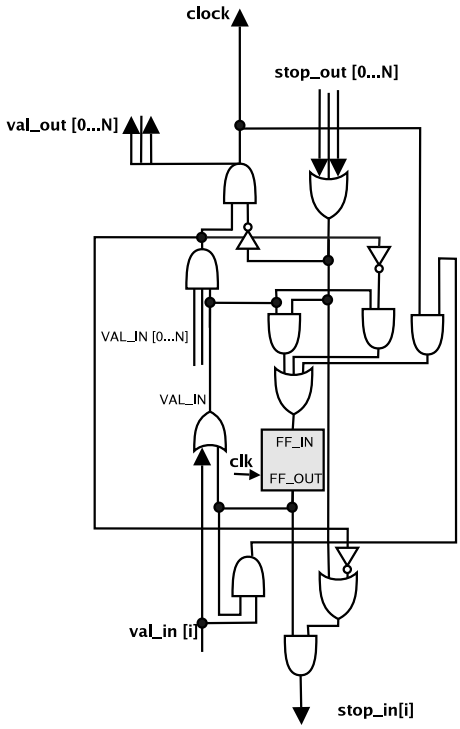


Figure 7: Shell - Circuit

The first property can be checked by the observer in figure 6.

The check will then consist in proving that such states are unreachable in all RSs. The second property could be modeled in a restricted case by *"tagging"* the successive data signals with indices, and then checking that these indexes are returned by the line in the same order as they were entered in the other end. The simplest scheme is to alternate 0 and 1 tags, providing an *alternated bit protocol* type verification.

We checked these properties by model-checking, with (low-range) constants replacing the integer parameters, and observers built from these formulas.

4 Shell wrappers

4.1 Shell modeling

Here our model follows rather closely the one of Casu and Macchiarulo [25]. It is depicted in figure 7.

Shell equations

- $stop\_in_i = (\bigvee stop\_out_{0...N} \bigvee \neg VAL\_IN_{0..N_3}) \bigwedge flipflop_{out}$.
- $VAL\_IN_i = val\_in_i \bigvee flipflop_{out}$.
- $val\_out_{0...N_1} = clock = \neg \bigvee stop\_out_{0...N_2} \bigwedge VAL\_IN_{0...N_3}$.
- $flipflop_{in} = (VAL\_IN_i \bigwedge stop\_out_{0...N_2}) \bigvee$
  $(VAL\_IN_i \bigwedge \neg VAL\_IN_{0...N_3}) \bigvee$
  $(val\_in_i \bigwedge flipflop_{out} \bigwedge clock)$.

As mentioned in section 2, one can consider the case where shells and pearls have potential zero-delay propagation (as long as there is no combinatorial loop involving only shells, without crossing a relay station). The shells will need the ability to store data that have already arrived, awaiting others still missing.

The Shell works as follows:

- The internal pearl's $clock$ and all val_out_i valid output signals are generated once we have all val_in, while $stop$ is false. The internal $stop$ signal itself represents the disjunction of all incoming $stop\_out_j$ signals from out-coming channels.
- the buffering register of a given input channel is used meanwhile as long as not all other input data are available.
- so, internal pearl's $clock$ is set to false whenever a backward $stop\_out_j$ occurs as true, or a forward val_in_i is false. In such case the registers already busy hold their *true* value, while others may receive a valid data *"just now"*.
- $stop\_in_i$ signals are raised towards all channels whose corresponding register was already loaded (a data was received before, and still not consumed), to warn them not to propagate any value in this clock cycle. Of course such signal cannot be sent in case the data is currently received, as it would raise a causality paradox (and a combinatorial cycle).
- flip-flop registers are reset when the pearl's clock is raised, following it to compute for one step and to consume its input data. The signal $stop\_in_i$ is raised only when the $flipflop_i$ is already holding a value and a $stop\_out$ is raised.

We should remember the constraint demanded by the relay stations for proper functioning, namely that on each output channel from the producer (is this case the shell), one has $stop\_out_j \Rightarrow \neg val\_out_j$, which holds here.

4.2 Correctness properties and formal verification

Keeping in mind relay stations, we want to show this property:

- data cannot be accepted before the previous one is processed: data order is preserved.

The property can be checked simply because the shell is connected *synchronously* to a relay station (or another shell) and thus the relay station cannot send any data to the shell when the shell is holding a data. The shell can have only one datum from each channel as said before then it cannot overwrite or loose this data until all needed datum are present to react. The data order is preserved, because by hypothesis the interconnection network is only point to point, cannot loose data or alter data ordering, the shell is waiting for all datum and then react, thus partial order of the desynchronized design is compatible with the synchronous one. We can also apply the alternated bit protocol verification in this case. The Shell is dead-lock free because we already established it as PN markings conditions. The model has been built with Esterel and formally verified with the modelchecker Xeve [9] and the visual tool Autograph [26]. Xeve is a BDD-based modelchecker which can, in addition of proving properties, provide a canonical minimal automaton for the component (up to bisimulation). Autograph allows the user to display graphically this minimal automaton (and recognize its characteristics, when the state number remain manageable).

5 Further topics

So far the theory developed here only consider the case where local synchronous components all consume and produce data on all input and output channels in each computation step, and where they all run on the same clock. In this favorable case functional determinacy and confluence are guaranteed, with latencies only impacting the relative ordering of behaviors. So it can be proved that the relaxed-synchronous version produces the same output streams from the same input streams as the fully synchronous specification (indeed the rank of a data in a stream corresponds to its time in the synchronous model, thereby reconstructing the structure of successive instants). Several papers considered extensions in the context of GALS systems, but then ignored the issue of functional correspondence with an initial well-clocked specification, which is our important correctness criterion.

This strong assumption can be weakened in a number of ways. Some are related to the various relative speeds and cadences of components in clock cycle rates, some are borne in the extension of Marked/Event graphs to more general subclasses in Petri nets in the asynchronous setting, and the most important ones are linking the two.

This concern is reflected elsewhere in the Ptolemy environment [19], where the so called SDF [18] domain corresponds to a slight extension of Event Graphs, whereas potential conflict choices are introduced in more general BDF and DDF domains [10] with the inherent problem that static bounded FIFO scheduling becomes undecidable.

One can extend the framework by allowing different cadences (so that various processing blocks run at different speeds, expressed as integer multiples of the master clock). More generally, each component can be assigned its own clock, with the assumption that all clocks are subclocks of a master clock, but not necessarily periodic. One can then build multi-rate/multi-clock systems. But, unless global rates are perfectly equalized around each loops, this might require fact component with different clocks be fed streams of data of unequal lengths. Usually the link with a fully synchronous specification is attempted by introducing a specific *absent* value for every interconnection signal, so that subclocks are defined as ticking only during the instants where a given triggering signal is *not* absent.

In general PN theory a place can be supplied tokens (here abstracting the data put in a FIFO channel) from various transitions (here processing elements). It thus merges the two flows (as a *mux*). The place can also offer its tokens at the other end to various consumers, thus operating a fork (or a *demux*) of the data flow carried through the channel. In other words tokens are shared. It gets difficult then to imagine that the rank of a data in a channel stream will recall the instant it was exchanged in a fully synchronous specification. Still, one can design a "locally-synchronous" version of places (we consider here the case of two producers and two consumers to this place): it has a main running clock, and two subclocks (one for input and one for output), so that data are taken from one input channel when the input subclock is raised, from the other otherwise (and similarly for output).

Of course the two kinds of extensions are linked, since channel sharing imposes that multiple productions or consumptions do not clash, so that it can be established that they are mutually exclusive (by being driven on exclusive subclocks). The issue of success is to guarantee liveness and throughput in the global system. This should be attained by devising the proper scheduling, which should generate the clock pulses at proper rates (in latency and cadence), so that data flow in the system smoothly. Several steps exist in this direction, with the notion of multiclock systems and clock calculus in synchronous languages [1, 8].The correctness criterion is that no component should ever require the presence of a signal data that is absent, and that signal data are not inappropriately lost (sometimes it is ok to ignore and discard them). Studies were also conducted to as when the seemingly monolithic synchronous specification in fact exhibited asynchronous behaviors based on independent clocks underneath [27].

Finally, the goal would be to define a general GALS modeling framework, where GALS components could be put in GALS networks (to this day the framework is not compositional in the sense that local components need to be synchronous). A system would consist again of computation and interconnect communication blocks, this time each with appropriate triggering clocks, and of a scheduler providing the subclocks computation mechanism, based on their outer main clock and several signals carrying information on control flow.

Our attention was brought by an anonymous referee to some recent work [29, 28], where relay stations are disposed of and replaced by parallel lines which carry the data alternatively in a round-robin fashion. More work is needed to compare our approach with the formal model provided there.

Finally, it was often suggested in previous papers that *latency equalization* could be a solution direction; the intent is to add extra (non physical) latencies to ensure that all proper input data are provided *simultaneously* to the local computing block. The *"non-physical"* new latencies can then be shifted up and down the network, under some semantic-preserving constraint, to optimize the global cycle allocation of computation activities. They can even be used to allow resynthesis of components under less stringent constraints. Nevertheless, preliminary enquiries showed us that the property of "equalizability" is far from obvious to characterize, and there are simple examples of networks where latencies can indeed *not* be evenly leveled. More investigations on this interesting criterion are in order.

References

[1] P. Amagbedon, L. Besnard, and P. Le Guernic. Implementation of the Data-flow synchronous language Signal. In *Proceedings PLDI'95*, 1995.

[2] C. André. Structural transformations given B-Equivalent PT-Nets. In *Application and Theory of Petri Nets*, 1981.

[3] C. André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *Computational Engineering in Systems Applications*, pages 19–29, 1996.

[4] C. André. Semantics of S.S.M. Technical report, I3S, CNRS, Esterel Technologies, 2003.

[5] F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[6] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In *Proceedings CONCUR'99*, volume 1664 of *LNCS*, 1999.

[7] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] G. Berry and E. Sentovich. Multiclock Esterel. In *Proceedings CHARME'01*, volume 2144 of *LNCS*, 2001.

[9] Amar Bouali. Xeve, an ESTEREL Verifi cation Environment. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 500–504, London, UK, 1998. Springer-Verlag.

[10] J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993.

[11] Luca P. Carloni and Alberto Sangiovanni-Vincentelli. Combining Retiming and Recycling to Optimize the Performance of Synchronous Circuit. In *The Proceedings of the 16th Symposium on Integrated Circuits and System Design*, 2003.

[12] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Design Automation Conference*, pages 361–367, 2000.

[13] Mario R. Casu and Luca Macchiarulo. A New Approach to Latency Insensitive Design. In *DAC'2004*, 2004.

[14] Ajanta Chakraborty and Mark R. Greenstreet. A Minimalist Source-Synchronous Interface. In *Proceedings of the 15th IEEE ASIC/SOC Conference*, pages 443–447, September 2002.

[15] Tiberiu Chelcea and Steven M. Nowick. Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols. In *Design Automation Conference*, pages 21–26, 2001.

[16] F. Commoner, Anatol W.Holt, Shimon Even, and Amir Pnueli. Marked Directed Graph. *Journal of Computer and System Sciences*, 5:511–523, october 1971.

[17] J. Cortadella, A. Kontratyev, L. Lavagno, and C. P. Sotiriou. A Concurrent Model for De-Synchronization. In *12th International Workshop on Logic and Synthesis*, 2003.

[18] E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. In *Proc. of IEEE*, 1987.

[19] Edward A. Lee. Overview of ptolemy project. Technical report, University of California, Berkeley, July 2003.

[20] C.E. Leiserson and J.B. Saxe. Retiming Synchronous Circuits. *Algorithmica*, 6, 1991.

[21] Luca P.Carloni, Kenneth L.McMillan, and Alberto L.Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[22] Luca P.Carloni, Kenneth L.McMillan, Alexander Saldanha, and Alberto L.Sangiovanni-Vincentelli. A Methodology for Correct-by-Construction Latency Insensitive Design. In *THE BEST OF ICAD*, 1999.

[23] Dumitru Potop-Butucaru, Benoît Caillaud, and Albert Benveniste. Concurrency in synchronous systems. In *Proceedings ACSD'04*, 2004.

[24] François R.Boyer, El Mostapha Aboulhamid, Yvon Savaria, and Michel Boyer. Optimal Design of Synchronous Circuits Using Software Pipelining. In *Proceedings of the ICCD'98*, 1998.

[25] Mario R.Casu and Luca Macchiarulo. A Detailed Implementation of Latency Insensitive Protocols. In *FMGALS 2003 Proceedings*, 2003.

[26] Valérie Roy and Robert de Simone. Auto/autograph. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 65–75, London, UK, 1991. Springer-Verlag.

[27] Montek Singh and Michael Theobald. Generalized Latency-Insensitive Systems for Single-Clock and Multi-Clock Architectures. In *DATE'04*, 2004.

[28] Syed Suhaib, David Berner, Deepak Mathaikutty, Jean-Pierre Talpin, and Sandeep Shukla. A functional programming framework for latency insensitive protocol validation. Technical report, Virginia Tech, March 2005.

[29] Syed Suhaib, David Berner, Deepak Mathaikutty, Jean-Pierre Talpin, and Sandeep Shukla. Presentation and formal verifi cation of a family of protocols for latency insensitive design. Technical report, Virginia Tech, February 2005.