# Interface Design for Rationally Clocked GALS Systems

Joycee Mekie[1] Supratik Chakraborty[1] Girish Venkataramani[2] P. S. Thiagarajan[3]
D.K. Sharma[1]
[1] Indian Institute of Technology, Bombay [2] Carnegie Mellon University
[3] National University of Singapore

## Abstract

*We investigate the problem of designing interface circuits for rationally clocked modules in GALS systems. As a key contribution, we show that knowledge of flow-control protocols can be used to significantly optimize synchronization mechanisms. We present delay-augmented netcharts as a formalism for representing communication protocols and describe techniques to analyze them. We use the results of our analysis to design a simple yet generic interface that is optimized for the given protocol and is free from synchronization failures. We show by means of case studies the inherent advantages of our methodology over an existing solution technique.*

## 1 Introduction

We investigate the problem of designing interfaces for communication between rationally clocked modules in Globally Asynchronous Locally Synchronous (GALS) systems. There are two fundamental issues that need to be addressed in designing such interfaces: *flow-control* and *synchronization*. Flow control is needed to ensure that the average rates of sending and receiving data are matched. This prevents overflow of interface data buffers. Synchronization ensures that all data sent by the sender arrive outside the setup-hold window of the receiver's clock. This avoids metastability-induced sampling errors. Earlier work on interface design for multi-clocked systems either address these two problems independently, or simultaneously through a common circuit design. Unfortunately, solutions that use independent mechanisms for flow-control and synchronization do not attempt to optimize one mechanism from knowledge of the other. Solutions that use the same circuit to address both concerns simultaneously often impose stringent constraints on the system, resulting in degraded system performance. In this paper, we argue that while flow-control and synchronization are orthogonal concerns, their individual solutions can be combined to obtain an optimized interface free from both problems. Though we choose to focus on the synchronization problem, we show that, unlike previous work, synchronization circuits can be optimized significantly by using the knowledge of inter-module communication protocols used to enforce flow-control. We present a formalism for specifying a class of communication protocols, and describe a methodology for designing interface circuits that are optimized for a given protocol, are free of synchronization errors, and minimize performance penalties.

We visualize a GALS system as a set of locally synchronous modules (henceforth called *LS modules*), each of which alternates between phases of computation and communication. The LS modules are assumed to be clocked rationally and phase-locked periodically. Thus, for every pair of modules $M_i$ and $M_j$, if the clocks driving these modules have periods $T_i$ and $T_j$ respectively, there exist positive integers $m$ and $n$ such that $\frac{T_i}{T_j} = \frac{m}{n}$. In addition, the phases of all clocks are aligned once every $T$ units of time, where $T$ denotes the *least common multiple* (*lcm*) of all $T_i$'s. We assume that all modules start functioning from a designated zero time, with all clocks aligned in phase. The modules communicate pairwise through pre-defined protocols that involve exchange of data and/or control messages. We assume that flow-control is achieved through these protocols and through buffering schemes. If off-the-shelf IP cores are used to build a GALS system, the expected sequences of events at the interfaces of communicating modules may not be compatible. We assume, through a separation of concerns, that all such incompatibilities have been resolved by using suitable *protocol converters* [9]. Thus, in the present setting, communicating modules will be assumed to agree on the sequence of events at their interface.

We represent the sequence of computation and communication actions for various LS modules in a GALS system by a restricted type of *netcharts* augmented with delay annotations on places. Netcharts [8] provide a clean way of representing unbounded sequences of non-atomic interactions between a finite set of processes. Augmented with delay annotations, they also permit the representation of arbitrary delays, attributable to computation, between successive communications.

Given a netchart representation of the inter-module communication protocol and sizes of all interface buffers, we describe how to check for the absence of buffer overflows – a key requirement of flow-control. We then present a technique to analyze the protocol and discover relations between the time of sending a message from one clock domain and the time of its sampling in another domain. The key requirement here is that these relations must be valid for *all* patterns of interaction between the *LS* modules that are consistent with the protocol. We then describe an interface design methodology that makes use of these timing relations to eliminate synchronization failures while minimizing performance loss. Our interface design is simple yet generic enough to be applicable to all handshake based communication protocols between rationally clocked modules. For convenience, we focus here on GALS systems in which each interaction involves exchange of messages between just two *LS* modules. Furthermore, as a first step, we restrict our attention to systems in which the top-level control flow for each component is *cyclic*. We note however that this cyclic requirement applies just to the pattern of interactions that each LS module goes through. The computations that an LS module executes, which we model via delay annotations, may well have complex control flows.

The remainder of this paper is organized as follows. In Section 2, we discuss the orthogonality of flow-control and synchronization issues. In Section 3 we present a brief overview of earlier work on interface design of multi-clocked systems. Section 4 presents delay-augmented netcharts, and describes a method for checking the absence of buffer overflows in a communication protocol specified by a netchart. A technique for analyzing such netcharts to discover relations between the sending and receiving times of messages is also presented in this section. Section 5 describes how these relations can be used to design an optimized interface circuit that guarantees freedom from synchronization errors. A few case-studies, comparing the advantages of our approach with earlier work, are presented in Section 6. Finally, we conclude the paper in Section 7.

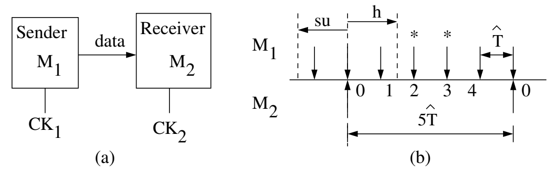


Figure 1. Flow-control and synchronization

## 2 Flow-control and synchronization

In this section, we argue that flow-control and synchronization are orthogonal concerns and hence their separation can be exploited in the design of interface circuits.

To illustrate the orthogonality, let us consider the system shown in Fig. 1(a). Suppose the clock periods of the sender and receiver are $\widehat{T}$ and $5\widehat{T}$ respectively. Suppose further, that the first ticks of the sender and receiver clocks are synchronized, and subsequently, the clocks synchronize once every $5\widehat{T}$. If the receiver's setup and hold windows are each $1.5\widehat{T}$ wide, and if the delays of the wires from the sender to the receiver are negligible, then there are no synchronization failures if the sender sends data items every $(5k+2)\widehat{T}$ and $(5k+3)\widehat{T}$, for all integers $k \geq 0$. This situation is depicted in Fig. 1(b), where data is sent only on clock ticks labeled *. However, if the receiver can receive only one data item every one of its clock's ticks, we have a flow-control problem, since the rate of sending data exceeds that of receiving it. Any finite interface buffer is therefore bound to overflow. Consider now the alternative scenario where the sender sends a data item every $(5k+4)\widehat{T}$ for all integers $k \geq 0$. Thus, every send of a data item is followed by at least one clock tick of the receiver. The rates of sending and receiving data are indeed matched and there are no flow-control problem. Unfortunately, the setup time of the receiver is violated every time the receiver samples data sent by the sender, leading to repeated synchronization failures. Thus flow-control and synchronization are orthogonal problems, and one is not necessarily eliminated by eliminating the other.

We now show that individual solutions to the flow-control and synchronization problems can be effectively combined to obtain an interface that is free from both problems. Consider the system shown in Fig. 2(a). Let $T_1$ and $T_2$ be the rationally related clock periods of modules $M_1$ and $M_2$, respectively. We assume that flow-control is achieved by means of (i) a communication protocol that limits the difference between the number of data items sent by one module and that received by the other to a pre-defined number $n$ $(> 0)$, and (ii) buffering circuits that can buffer up to $n$ data

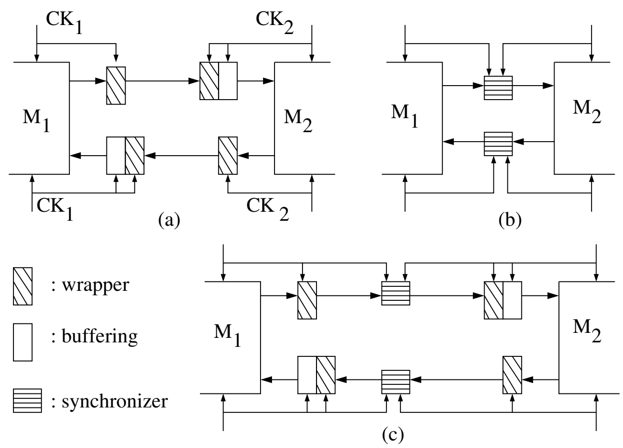


Figure 2. Composing solutions

items, and present data to module $M_i, (i \in \{1, 2\})$ at a rate no greater than one data item per $T_i$ units of time. As shown in Fig. 2(a), an implementation of this mechanism requires the possible addition of wrappers around each module to ensure that they conform to the protocol and the use of buffering circuits which may be merged with the wrapper on either the sender's or receiver's end. In the absence of synchronization errors at the local interfaces of the LS modules, the above mechanism suffices to guarantee error-free communication between the modules. Suppose now we also have synchronizing circuits that receive data items from a sender module and ensure that (i) each data item is presented to the receiver at a time that meets the setup and hold times of the receiver's clock, and (ii) all data items are presented to the receiver in the order in which they are received from the sender. Thus, in the absence of flow-control problems, synchronizing circuits suffice to ensure error-free communication between a pair of modules, as shown in Fig. 2(b). In general, the design of a synchronizing circuit must take into account the rational relation between the sender's and receiver's clocks as also the setup and hold windows of the receiver. It is therefore not uncommon to have synchronizing circuits that take the sender's and/or receiver's clocks as inputs, as depicted in Fig. 2(b).

We now combine the flow-control and synchronization solutions as shown in Fig. 2(c) to obtain an interface circuit that is free from both problems. However, for this ensemble to work correctly, it is necessary to fine-tune the synchronizing circuit to account for the wrapper and buffering delays. Thus the setup-hold window of the receiver module in Fig. 2(c) must be shifted by the delay of the wrapper and buffering circuits to determine the effective forbidden window at the output of the synchronizing circuit. The wrappers and buffering circuits, of course, continue to enforce flow-control regardless of the presence of the synchronizing circuits. Thus the resulting interface is free from both synchronization and flow-control problems (assuming perfect synchronizers). One may also consider placing the synchronizing circuits between the output of the buffering circuits and the receiver. However, we will focus on the configuration shown in Fig. 2(c).

# 3 Related work

As indicated in Section 1, earlier work on interface design for multi-clocked systems either address the flow-control and synchronization problems independently or use a single circuit to achieve both ends. Synchronization is usually achieved by one of several means. This includes the use of synchronizers [11], mechanisms to ensure that the sender sends data only in pre-determined *safe* cycles [10], adjusting delays between the sender and receiver [2, 5] and even stretching the receiver's clock to avoid setup and hold time violations [12]. The flow-control problem is typically addressed through handshaking protocols [9] or by stalling the sender at appropriate times. The latter is usually achieved either by control signaling [10, 1] or by temporarily stopping the sender's clock [12, 6]. Unfortunately, solutions that involve stopping or stretching of clocks are not well-suited for high-speed designs with IP cores having large clock-buffer delays [6, 4]. Consequently, practical designers often avoid this solution.

A common feature of the various existing solutions is that they either de-couple the synchronization and flow-control mechanisms completely, or couple them very strongly. Solutions in the former category do not attempt to optimize one mechanism using knowledge of the other. Solutions in the latter category impose stringent constraints on the operation of the system. For example, they may force the use of a *specific* flow-control protocol that is tightly coupled with the implementation of the synchronization mechanism [5, 10, 12]. This prevents the designer from choosing a flow-control protocol that best suits a particular application. It also impedes using information about the protocol to optimize the synchronization mechanism. Our work fills in this gap for rationally clocked modules.

The work that comes closest to our is that of Sarmenta et al [10]. In their approach, the synchronization and flow-control mechanisms are strongly coupled and the interface designer has little choice in selecting a communication protocol that suits the given application and also achieves flow-control. The key idea behind Sarmenta et al's solution is to statically analyze the rational relations between sender and receiver clocks and identify sender clock cycles that are *unsafe* for sending data for both flow-control and synchronization reasons. Their interface design then explicitly prevents the sender from sending data in these cycles by control signaling. This approach completely ignores the actual protocol used to communicate between the sender and receiver modules. Thus, it is a worst-case design, catering to worst-case communication protocols that cause maximum synchronization and flow-control problems. This can lead to performance penalties in a system that does not employ such worst-case protocols.

In contrast to Sarmenta et al's approach, we assume the presence of a global communication protocol that, along with suitable buffering schemes, enforces flow-control. We then use our knowledge of this protocol and the rational clock relations to determine a strategy for correcting possible synchronization errors in each cycle in which the sender wishes to send a data item. We show by means of case studies that this approach results in significantly simpler interfaces than the worst-case interface designs of Sarmenta et al.

## 4 Delay-augmented netcharts

We now present a formalism for expressing a large class of communication protocols between rationally clocked modules in GALS systems. We also describe techniques for analyzing protocols represented in this formalism.

Our formalism is based on netcharts introduced by Mukund et al [8]. A netchart consists of a two-level representation of regular sequences of non-atomic interactions between a finite set of processes. At the top level, a suitably structured Petri net is used to depict the control flow in a network of sequential processes. At the second level, the (synchronization) transitions involving a set of processes are refined into Message Sequence Charts (MSCs) with a key restriction: if transition $t$ is refined into the chart $Ch_t$ then the life-lines of processes taking part in $Ch_t$ must be precisely the set of processes taking part in $t$ at the top level description. Throughout this paper, we will make two simplifying assumptions : (i) each MSC represents the sending of a *single* message or data item from one process and its receipt by another process, and (ii) the top-level control flow for each process is *cyclic*. Our work also applies to complex MSCs that can be decomposed into atomic MSCs satisfying the first restriction. We use these atomic MSCs since they *localize* the interactions between the components. In contrast, if we use the standard model of sequential processes communicating via point-to-point buffers, the information concerning *from where* a process sent a message and

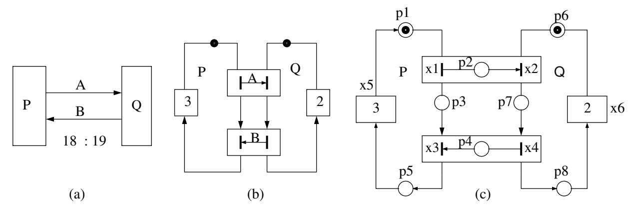


Figure 3. (a) System (b) Netchart (c) Petri net

*at where* it was received will be lost. This information is crucial for designing the required interface circuits. As remarked in the introduction, the internal computations can have complex control flows. All we require is that they should have well-defined beginnings and endings. In the subsequent discussion, we will refer to modules and processes interchangeably in the context of netchart representations of communication between rationally clocked modules. We omit a detailed introduction to netcharts and instead refer the reader to [8] for further details.

We assume that the top-level cyclic control flow for each module consists of a sequence of *communication* and *computation* transitions. A communication transition is one that refines to an MSC representing the transfer of a message between modules. A computation transition is one that has exactly one incoming and one outgoing place. The corresponding MSC has only one life-line – that of the process representing the module. We augment the basic netchart representation by annotating computation transitions with positive integers representing the number of cycles in terms of its clock-period required by the corresponding module to complete the computation. An annotation of $\Delta^+$ on a computation transition indicates that the corresponding computation takes a non-deterministic number of clock cycles, ranging from 1 to $\infty$. We call such annotated netcharts as *delay-augmented netcharts*. The sending of a message or the process of receiving a message is always assumed to take exactly one clock cycle of the corresponding module. However, multiple cycles may elapse after a message is sent by one module and before it is picked up (sampled) by the designated receiver. Thus, each unidirectional channel of communication between a pair of modules must in general have buffering capabilities.

To see how netcharts are used to represent sequences of computation and communication, let us consider the example shown in Fig. 3(a). Here, $P$ and $Q$ are two rationally clocked modules that engage in computation and communication actions in the following sequence. Starting from a designated zero of time with the clocks of both $P$ and $Q$ aligned, $P$ first sends message $A$ to $Q$ and then waits to receive message $B$ from $Q$. Module

$Q$, on receipt of message $A$ proceeds to send message $B$ to $P$ and then engages in a computation that takes 2 cycles of $Q$'s clock. After finishing this computation, $Q$ restarts its sequence of operations, waiting to receive message $A$ from $P$. Module $P$, on receipt of message $B$ from $Q$, proceeds to compute for 3 cycles of $P$'s clock. In the subsequent cycle, it restarts its sequence of actions by sending message $A$ to $Q$ again. The initial token marking in the netchart has a connotation similar to that of initial token markings in a Petri net. Thus, the initial marking determines the communication or computation action that each module engages in at the designated zero of time. Using the simple technique in [8], Fig. 3(c) shows the underlying Petri net for the netchart depicted in Fig. 3(b). It can be seen from this Petri net that transition $x2$ cannot fire unless both places $p6$ and $p2$ have tokens. This captures the requirement that $Q$ cannot receive a message unless it is ready to receive and a new message is available from $P$.

### 4.1 Checking for buffer overflows

Given a delay-augmented netchart representing the sequence of actions of a set of modules, the rational relations between all module clocks, and the sizes of buffers in all unidirectional communication channels, we now describe a method to check if any sequence of actions consistent with the protocol can cause a buffer to overflow.

Let $N$ be a delay-augmented netchart, and let $T_1, T_2 \ldots T_n$ denote the set of rational clock periods of modules $M_1, M_2 \ldots M_n$, respectively, participating in the protocol represented by $N$. Let $BasicPN(N)$ denote the underlying Petri net of netchart $N$, obtained by expanding each MSC action, as explained by Mukund et al [8] and illustrated in Fig. 3(c). For each module $M_i$, we use $C_i$ to denote the simple cyclic component of $BasicPN(N)$ representing the sequence of actions (computation and communication) of $M_i$. If $\widehat{T}$ denotes the *greatest common divisor* (henceforth called *gcd*) of $\{T_1, \ldots T_n\}$, we use $k_i$ to denote the ratio $\frac{T_i}{\widehat{T}}$, for all $i$ in 1 through $n$.

To check for buffer overflows, we first construct $BasicPN(N)$ from the given netchart $N$, and then apply the following sequence of transformations to each cyclic component $C_i$ of $BasicPN(N)$.

1. Each computation transition annotated by $\Delta^+$ is replaced by two computation transitions, each annotated by a delay of 1, as shown in Fig. 4(a).

2. For each transition $t$ that represents receipt of a message by $M_i$, a computation transition annotated by a delay of 1 is added as shown in Fig. 4(b). This models the waiting of the receiver for an additional clock cycle if it is ready to receive a message in the current cycle, but is unable to do so because the corresponding communication channel has no messages. Additional cycles introduced in the above steps will be called *auxiliary cycles* connected to $C_i$.

3. Each computation transition annotated by a positive integer delay $r$ is replaced by $r$ consecutive computation transitions, each annotated by a delay of 1. Dummy places are inserted between every pair of consecutive transitions.

4. Each auxiliary cycle connected to $C_i$ is replaced by a loop of $k_i$ places, with the place where the auxiliary cycle connects to $C_i$ remaining unchanged. Dummy transitions are then added in the expanded auxiliary cycle between every pair of consecutive places.

5. Finally, each place in the (possibly) expanded cycle $C_i$ of the transformed Petri net is replaced by a sequence of $k_i$ new places. Dummy transitions are inserted between every pair of consecutive places. If a place had an initial token in $BasicPN(N)$ or if it was connected to an auxiliary cycle, then the last of the $k_i$ new places that replaces this place inherits the initial token or the connection to the auxiliary cycle, respectively.

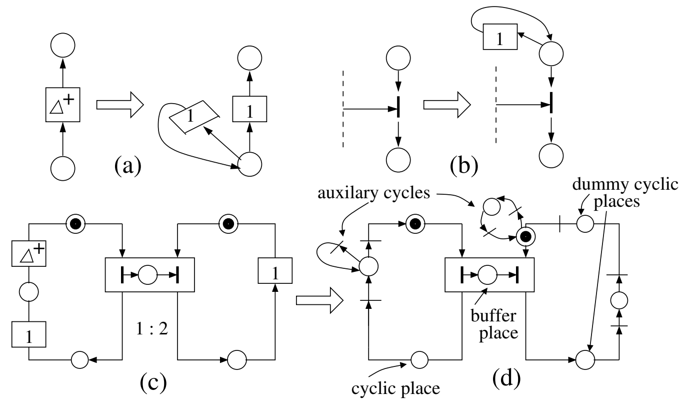


Figure 4. Building an expanded Petri net

The above transformations are illustrated in Fig. 4. We will call the Petri net obtained after applying the above transformations to all cyclic components in $BasicPN(N)$ as $ExpandedPN(N)$. Fig. 4(d) shows the $ExpandedPN(N)$ for the $BasicPN(N)$ in Fig. 4(c). We will henceforth refer to the expanded cycle $C_i$ as well as all auxiliary cycles connected to $C_i$ as *expanded cycles* of module $M_i$.

The construction above ensures that each transition in $ExpandedPN(N)$ belongs exactly to one expanded cycle. In addition, each place in $ExpandedPN(N)$ either belongs to an expanded cycle of some module, or its immediate predecessor is a transition representing the sending of a message by module $M_i$, and its immediate successor is a transition representing the receipt of the message by another module $M_j$. We will call places of the first type as *cyclic places*, and places of the second type as *buffer places*. Let $n_{i,j}$ denote the buffer size associated with each unidirectional communication channel from module $M_i$ to $M_j$. The buffer overflow problem can now be checked by exploring the state space of $ExpandedPN(N)$ with the following restrictions. Firstly, a state transition is effected if and only if every token in a cyclic place moves to a successor cyclic place. Secondly, a transition representing the receipt of a message must fire if both its predecessor places have tokens. Finally, whenever a buffer place corresponding to the sending of a message from $M_i$ to $M_j$ contains $n_{i,j} + 1$ tokens, we stop the state space exploration and report a buffer overflow in the unidirectional channel from $M_i$ to $M_j$. It can be shown that the above state space exploration always terminates, given finite values of all $n_{i,j}$. More details including a crude upper bound on the size of the state space can be found in [7].

## 4.2 Detecting message send/receive times

Once a netchart has been verified to be free of buffer overflows, the state space exploration of $ExpandedPN(N)$ can be augmented to yield valuable information relating the time when a message is sent from one clock domain and the time when it is correctly sampled in another clock domain. Using the notation of the previous subsection, we let $T$ denote the *lcm* of $T_1$ through $T_n$ and will express all timing relations in this section modulo $T$. Furthermore, since each clock period is an integral multiple of $\widehat{T}$, we can talk of timing relations between clocks in multiples of $\widehat{T}$. In the subsequent discussion, we will use $L$ to denote the ratio $\frac{T}{\widehat{T}}$. We will also assume that the clock period of every module exceeds the setup-hold window of sampling elements in that module.

Let $K$ and $P$ denote the the total number of cyclic and buffer places, respectively, in $ExpandedPN(N)$. We define an *augmented state* of $ExpandedPN(N)$ to be a 3-tuple $(X, Y, Z)$, where $X$ is an integer in the range 0 through $L - 1$, $Y$ is a $K$-bit binary vector, and $Z$ is a $P$-tuple of pairs of integers. We use $X$ to remember the number of ticks of the base clock (i.e. one with period $\widehat{T}$) modulo $L$ that are needed to reach the current augmented state. We use the $r^{th}$ bit of $Y$ to remember whether the $r^{th}$ cyclic place contains a token in the current augmented state. Finally, if the $r^{th}$ pair in $Z$ is $(\alpha_r, \beta_r)$, it represents the following information about the $r^{th}$ buffer place in $ExpandedPN(N)$. The component $\alpha_r$ stores the number of tokens in this buffer place (representing the number of messages stored in the corresponding interface buffer) in the current augmented state. If $\alpha_r > 1$, the interface buffer has at least one message. However, it may not be possible for the receiver to correctly sample this in its next clock tick due to setup-time violation. Therefore, we use $\beta_r$ to remember the additional time needed after entering the current augmented state and before the setup constraint for correct sampling of the first message in the buffer is met. The information in $\beta_r$ is crucial to determine when the subsequent actions of the receiver must happen. Indeed, if the sender sends a message at a time that violates the setup time of the next available receiver clock tick, the message must be delayed so that it is sampled correctly by the subsequent receiver clock tick. This, in turn, will delay all subsequent actions of the receiver by one cycle. Hence, keeping track of the time (modulo $T$) when the receiver engages in its computation and communication actions requires knowledge of when the receiver correctly samples incoming messages. If $SU_j$ denotes the setup time of the sampling elements in module $M_j$, and if the $r^{th}$ buffer place models buffering of messages sent from $M_i$ to $M_j$, we set $\beta_r$ to $SU_j$ whenever a token is inserted into the empty $r^{th}$ buffer place. Otherwise, if this buffer place already contains a token, we simply update $\beta_r$ to record the remaining time until the setup constraint of the first message in the buffer is met. Note that we only worry about the setup time violation of the first message that enters an empty buffer. We do not concern ourselves with additional messages, if any, in the buffer. This is because if the first message in a buffer containing multiple items is sampled correctly, the subsequent ones are guaranteed to be sampled correctly, since at least one clock period of the receiver must elapse between two consecutive samplings, and the clock period is assumed to be larger than the setup-hold window. Similarly, we do not worry about hold times since a data item that arrives within the hold window after a receiver clock tick can indeed be sampled correctly by the next clock tick of the receiver, assuming that the receiver clock period exceeds the setup-hold window.

We now explore the augmented state space of $ExpandedPN(N)$ with the following restrictions in addition to the restrictions outlined from the previous subsection: In the initial state, $X = 0$, $Y_i = 1$ iff the $i^{th}$ cyclic place has an initial token, and $Z_r = (0, 0)$ for all $r$ in 1 through $P$. When a state transition happens, $X$ is incremented modulo $L$, and all $Y_i$'s are updated to represent the presence or absence of tokens in the corresponding cyclic places. Similarly, $\alpha_r$ for all buffer places $r$ are updated to represent the number of tokens in $r$. Next, let $t$ be a transition in $ExpandedPN(N)$ corresponding to the receipt of a message by module $M_j$, and let $r$ be the buffer place immediately preceding this transition. Transition $t$ fires if and only if $\alpha_r > 0$, $\beta_r = 0$, and the cyclic predecessor place of $t$ has a token. Finally, for every buffer place $r$ that corresponds to the sending of a message from $M_i$ to $M_j$, $\beta_r$ is updated in the following manner: if $\alpha_r$ has changed from 0 to 1 as a result of the current state transition, $\beta_r$ is set to $SU_j$; else, if $\alpha_r$ is at least 1, $\beta_r$ is updated to $\max(\beta_r - \widehat{T}, 0)$.

The above restrictions ensure that the receiver waits an additional cycle to sample a message correctly, if there is a violation of setup time in the current cycle. They also ensure that the additional time needed to clear the setup window for a new data item is always updated correctly. It is easy to prove that the above exploration process always terminates [7].

Let $t$ be a transition representing the sending of a message from module $M_i$ to module $M_j$. Let $r$ be the cyclic place in $ExpandedPN(N)$ (in the expanded cycle corresponding to $M_i$) immediately following $t$. We now collect all augmented states $(X, Y, Z)$ that are discovered by the state space search and that have $Y_r$ equal to 1. These are states where module $M_i$ has just sent a message to $M_j$. For each such state, let $\tau$ denote the value of $X$. We then claim that $\tau.\widehat{T}$ corresponds to the time (modulo $T$) of sending a message from $M_i$ to $M_j$. Since all times are measured modulo $T$, one can now identify the clock tick of the receiver when this message can be correctly sampled. In particular, if the time from sending the message to the next receiver clock tick is less than $SU_j$, the message will be sampled by the receiver clock tick after the next. Otherwise, it will be sampled by the next receiver clock tick. Since the state space exploration finds all augmented states reachable by the system, by repeating the above process for all reachable states $(X, Y, Z)$ that have $Y_r = 1$, we can identify all time instants (modulo $T$) when a message is sent from $M_i$ to $M_j$. This allows us to obtain relations between the sending and receiving times of all messages in the given GALS system.

### 4.3 NuSMV Model

We have used the reachability analysis engine of NuSMV [3] to explore the augmented state space of $ExpandedPN(N)$, as described above. We outline below the construction of the NuSMV model from a netchart and rational clock relations. Details of our modeling technique and analysis can be found in [7].

Given a netchart and rational relations between all module clocks, the NuSMV model consists of: (a) a clock module that counts the number of base (or *gcd*) clock ticks modulo the *lcm* period $T$, (b) a module for each cyclic process in the netchart, and (c) a module for each unidirectional communication channel between two modules. We ensure that the module for the $i^{th}$ process engages in computation or communication actions only when the count of the clock module is an integral multiple of $k_i$ $(= \frac{T_i}{\widehat{T}})$. The module for a unidirectional communication channel between the $i^{th}$ and $j^{th}$ processes keeps track of the number of messages queued in this channel, and also the remaining time for the first message queued in the channel to meet its setup constraint. It does so by looking at the states of the modules for the $i^{th}$ and $j^{th}$ processes to determine whether a message is sent or received. All the modules are composed synchronously and buffer overflow checks and queries about message send times are posed as reachability (**EF**) queries in CTL to NuSMV.

As an example, consider the system shown in Fig. 3. The NuSMV model for this system consists of five modules : *clock*, $P$, $Q$, $PQ$, and $QP$ (the last two modeling unidirectional communication channels). Let the clock periods of $P$ and $Q$ be in the ratio 18 : 19; hence, the *gcd* $\widehat{T}$ of the periods is 1 and their *lcm* $T$ is 342. Let us assume that the sampling elements in both $P$ and $Q$ have setup time of $4\widehat{T}$ and negligible hold time. Using NuSMV, we can now verify that in this example, no buffer ever holds more than one message. In addition, messages are sent from $P$ to $Q$ only on the $0^{th}$, $126^{th}$, and $234^{th}$ ticks. (modulo 342, the *lcm* period) of the base clock. Similarly, messages are sent from $Q$ to $P$ only on the $38^{th}$, $152^{th}$ and $266^{th}$ ticks (modulo 342) of the base clock. Armed with this knowledge, we can now infer that that no messages are sent in either direction in *unsafe* cycles. In other words, no interface circuit is required between the modules, since their communication protocol and rational clock relations lead to no synchronization or flow control problems.

## 5 Interface Design

The analysis described in the previous section gives a set of time instants (modulo $T$) when a module can send a message to another module. We now wish to use this information to design an interface between these modules that is free of synchronization errors.

Let $SU$ and $H$ denote the setup and hold times respectively of sampling elements in the receiver module.

Assuming that the output of the sender is directly connected to the input of the receiver without any synchronizing circuit in between, let $C$ and $D$ represent the minimum and maximum delay, respectively, from the sender clock tick that triggers the dispatch of a message to the arrival of this message at the receiver. Let $\delta_{SH} = SU + H$ represent the setup-hold window and $\delta_{CD} = D - C$ represent the *contamination* or *uncertainty* window for sending a message. We assume that skew and jitter of all module clocks are small, and can be accounted for by shifting and/or widening the setup-hold window. We also assume that $\delta_{SH}$ and $\delta_{CD}$ are small compared to the clock periods of all modules. Although the effect of long interconnects has not been explicitly considered in this work, our analysis and design methodology applies as long as the above assumptions hold. As before, we let $T$ and $\hat{T}$ represent the *lcm* and *gcd* periods of the module clocks.

For each time instant $t_m$ (modulo $T$) identified by our analysis as the potential time of sending a message from one module to another, we first determine if the interval $[t_m + C, t_m + D]$ overlaps with the setup-hold window with respect to any receiver clock tick. If it does, messages sent at $t_m$ can indeed lead to synchronization errors. We propose to rectify this problem by selectively padding delays to messages sent at specific time instants, as described in the next subsection. We also describe a hardware implementation of this scheme, assuming an a priori bound of $n$ for the number of messages that can be queued up in a communication channel. Note that the analysis described in the previous section allows us to check if the bound $n$ is never exceeded during the execution of the system.

## 5.1 Selective insertion of delays

The idea of selectively delaying messages is best illustrated through an example. Let us consider again the system shown in Fig. 1(a) in which $M_1$ sends a message to $M_2$. Suppose, $M_1$ alternates between phases of communication and computation, and the number of cycles required for $M_1$'s computation phase varies non-deterministically from 1 to $\infty$. Therefore, $M_1$ can send a message to $M_2$ at any time instant $k\widehat{T}$ (modulo $T$), where $k \in \{0, 1, 2, 3, 4\}$. We assume that flow control is handled separately by the protocol so that interface buffers never overflow. Further, we assume that the delays $C$ and $D$ are negligible, and that each of $SU$ and $H$ are $1.5\widehat{T}$. From the clock phase relations shown in Fig. 1(b), it can then be seen that messages sent at times $k\widehat{T}$, where $k \in \{0, 1, 4\}$ can lead to synchronization errors. To rectify these errors, we need to delay these messages so that they arrive at the receiver outside the setup-hold window of the receiver's clock ticks. For example, a message sent at time corresponding to $k = 0$ must be delayed by at least $1.5\widehat{T}$. Similarly, a message sent at time corresponding to $k = 1$ must be delayed by at least $0.5\widehat{T}$, while one sent at time corresponding to $k = 4$ must be delayed by at least $(2.5\widehat{T})$. Messages sent at times corresponding to $k \in \{2, 3\}$ need not be delayed at all since they do not lead to any synchronization error. The above analysis gives the minimum delays that must be padded to messages sent at specific time instants to avoid synchronization errors. It is also useful to have upper bounds on the allowable delay paddings, so that the interface designer has greater flexibility in choosing delay elements in an implementation. For performance reasons, we must choose the upper delay bounds such that every message is sampled correctly and *as early as possible* by the receiver. Thus, in the example of Fig. 1, a message sent at time corresponding to $k = 0$ must not be padded with a delay greater than $3.5\widehat{T}$, if we wish to ensure that the message is sampled as early as possible while also avoiding synchronization errors. Thus, $[1.5\widehat{T}, 3.5\widehat{T}]$ gives the allowable range of delay padding for messages sent at time corresponding to $k = 0$. Similar delay ranges can be found for all other instants at which the sender can send a message.

### 5.1.1 Choosing appropriate delay elements

Let $T_1$ denote the clock period of the sender and $T_2$ denote the clock period of the receiver. In the worst case, we may need $\frac{lcm(T_1,T_2)}{T_1}$ delay elements in the above delay padding scheme – one for each $k \in \{0, 1, \ldots, \frac{lcm(T_1,T_2)}{T_1}\}$. However, it is often possible to reduce the number of delay elements required for a given interface. In order to do this, we must intersect all the delay ranges computed above. If the intersection is non-null, say $[\delta_a, \delta_b]$, and if a delay element in this range is available to the interface designer, this single delay element can be used for all $k \in \{0, 1, \ldots, \frac{lcm(T_1,T_2)}{T_1}\}$. We will call this a *1-delay (1D)* solution. However, if the intersection is empty, or if a delay element in the range $[\delta_a, \delta_b]$ is not available, we must partition the set of delay ranges into two sets, and find the intersection range for each partition. If these intersection ranges are non-empty, the designer must now choose two delays, one from each intersection range. A delay chosen from the intersection range of one partition can then be used for all delay ranges in that partition. We will call this a *2-delay (2D)* solution, and so on. As is obvious, this process must be continued until the designer is able to choose a delay element for each delay range identified by our above analysis.

An analysis of the communication protocol can help simplify interface design by pruning the set of time instants at which a message can be sent from one module to another. For example, it can be shown that if the modules in Fig. 1(a) communicate by means of a protocol that ensures that messages are sent only at times $k\widehat{T}$ where $k \in \{0, 1, 4\}$, a single delay element of $2.5\widehat{T}$ suffices to build a synchronizing solution.

### 5.2 Implementing selective delay padding

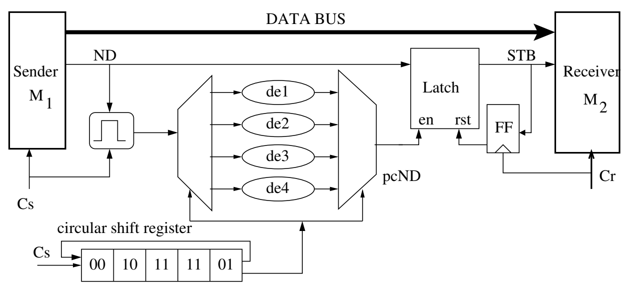


Figure 5. Selective delay padding

In System-on-Chip (SoC) designs employing GALS architecture, communication between disparately clocked modules is often implemented by sending data on a dedicated data bus, and by sending a control signal that indicates to the receiver that the data on the bus is valid. The selective delay padding scheme discussed above requires that each message be padded with an appropriate delay to ensure correct sampling by the receiver. If we insert delay elements on the data bus lines between communicating modules, the bus must then be switched between various delay elements, depending on the time instants at which messages are sent. This gives rise to an inefficient hardware implementation, since switching of wide data buses can consume significant power, and may also involves higher design complexity. We therefore propose padding appropriate delays to the control lines that are used to indicate the validity of data on the data bus.

As a typical example, consider a unidirectional communication channel between modules $M_1$ and $M_2$, as shown in Fig. 5. Let $T_1$ and $T_2$ be the rationally related clock periods of $M_1$ and $M_2$, respectively. To send a message to $M_2$, module $M_1$ raises the control signal $ND$ (new datum), and places a new data value on the output data bus. Module $M_2$ in turn checks if its $STB$ (strobe) input is high (active), and uses the data on the data bus whenever it finds $STB$ high. We ensure that $STB$ is maintained low when there is no valid data to be received by $M_2$. For multiple messages sent in consecutive cycles, we assume that the sender holds $ND$ high for multiple sender clock cycles, and pulls it low only when there are no further messages to be sent. To avoid synchronization errors, we need to ensure that $STB$ always arrives outside the setup-hold window with respect to the receiver's clock ticks.

The hardware implementation of our interface circuit is shown in Fig. 5. The output data bus from $M_1$ is directly fed to the input data port of $M_2$, while the delay padding is done on the control line. The delay selection circuit for the control signal $ND$ consists of suitably chosen delay elements, a multiplexer($MUX$), a demultiplexer($DMUX$) and a circular shift register clocked synchronously by $M_1$'s clock ($C_s$). The $MUX/DMUX$ are used to switch between different delay elements. The circular shift register supplies the select bits for the $MUX/DMUX$, and is pre-loaded with the sequence of bits required for choosing the appropriate delay element for each time instant when a message can be sent. The size of the circular shift register is $\frac{lcm(T_1,T_2)}{T_1}$. Since $ND$ can remain high for multiple clock cycles, we do not feed (delayed) $ND$ directly to the receiver. Instead, an $ND$-pulse generated from $ND$ and the sender's clock $C_s$ is padded with a suitable delay and corrected for phase. This phase corrected control signal *pcND* is used to *set* a latch, as shown in Fig. 5. The output of the latch feeds the $STB$ input of the receiver. We must reset $STB$ after the message has been received to prevent re-sampling and re-use of the same data by $M_2$ in the next receiver clock cycle. This can be done by using the acknowledgment, if available, from the receiver. As shown in Fig. 5, an additional flip-flop (FF) can also be used on the receiver's side to provide an acknowledgment if the receiver does not explicitly generate an acknowledgment. Note that the delays of the pulse generator, $MUX/DMUX$, shift register and latch must be taken into account when calculating $\delta_{CD}$ to find the time when a message sent from $M_1$ arrives at the interface of $M_2$. In addition, there are certain constraints between the delays of interface circuit components and the clock periods of communicating modules that must be met for the circuit in Fig. 5 to work correctly. These are detailed in [7], and are easily met if the module clock periods are larger than the cumulative delays of interface circuit components.

### 5.3 Buffering solution for flow control

The interface design described above works if at most one message needs to be buffered in every unidirectional communication channel. This design can be augmented with an $n-$stage circular FIFO at the receiver's end to handle at most $n$ ($\geq 1$) messages in each communication channel. We assume that $n$ is known a priori, and that the method of Section 4.1 has been used to ensure that no more than $n$ messages need to be buffered in the communication channel.

### 5.4 Comparison with Sarmenta *et al*'s scheme

We now compare our approach with that of Sarmenta *et al* [10] along the dimensions of (a) interface circuit overheads, and (b) impact on system performance. For ease of exposition, we will illustrate the advantages of our approach by means of simple examples.

**Circuit implementation overheads:** Let us consider the problem of designing an interface for two rationally clocked modules with clock periods in the ratio $5:6$. Let us assume that $SU = H = \widehat{T}$, $D = 3\widehat{T}$ and $\delta_{CD}$ is negligible. For these clock ratios and timing parameters, Sarmenta *et al*'s approach [10] involves the introduction of a *double-buffer* in the interface for achieving the maximum possible rate of message transfer between the communicating modules. The double buffering scheme requires additional data bus buffers, apart from look-up tables (LUTs), MUXes and DMUXes, and the hardware requirements of this approach are proportional to the width of the data bus. Moreover, the double buffering scheme requires a potentially wide data bus to be switched between two buffers, which leads to significant power consumption in the interface. Redundant switching of the data bus between the two buffers can be avoided on the sender's side by allowing the switching to happen only when there is a message to be sent. However, the data bus on the receiver's side must switch between the buffers in every receiver clock cycle to avoid synchronization errors. This is because Sarmenta *et al*'s approach does not analyze the communication protocol; hence the cycles in which data can potentially be received are not known a priori. Thus, the double buffering scheme is expensive in terms of both hardware and power.

Sarmenta *et al*'s approach is essentially a worst-case design approach, wherein a sender is assumed to send messages as frequently as possible. In real systems with known communication protocols, such worst-case communication patterns are however rare, and the protocol can often be analyzed to statically determine the cycles when a receiver can potentially receive messages. This can lead to significant optimization of the interface. For instance, if the communication protocol ensures that messages are sent only at such time instants

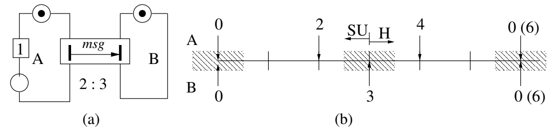


Figure 6. A and B with clock ratios 2 : 3

that do not lead to synchronization errors, the need for an interface circuit that avoids synchronization errors is completely obviated. We will see in Section 6 that such cases do arise in practice. Similarly, if the protocol ensures that no communication channel ever needs to buffer more than 1 message, no additional circuitry is required to enforce flow control.

**System performance:** In Sarmenta *et al*'s approach, a sender module is explicitly prohibited from sending messages on "unsafe" cycles that are statically determined independent of the communication protocol used by the communicating modules. Also, if the sender is slower than the receiver, their approach makes the worst-case assumption that the sender sends a message on every clock cycle. In reality however, the sender may send messages only on specific cycles as determined by the communication protocol. Thus, Sarmenta *et al*'s approach may prohibit the sender from sending messages in cycle $c_1$ and allow messages to be sent in a later cycle $c_2$, whereas the sender may indeed want to send a message on $c_1$ and not on $c_2$. This can cause the sender to be delayed longer than is necessary every time it wishes to send a message. and can lead to degradation in the system performance.

As an example, consider the communication protocol between modules $A$ and $B$ depicted in Fig. 6(a). Let us assume that the clock periods of $A$ and $B$ are in the ratio $2:3$ respectively, and that the *gcd* of the clock periods is $\widehat{T}$. Let us further assume that $SU = H = \frac{\widehat{T}}{2}$, $D = \frac{\widehat{T}}{8}$ and $\delta_{CD}$ is negligible. Fig. 6(b) shows the phase relations between the clock ticks of $A$ and $B$ within the *lcm* period. In Sarmenta *et al*'s approach, module $A$ is inhibited from sending messages at time instants $6m\widehat{T}$, where $m \geq 0$, since these are considered *unsafe* cycles for sending messages. Module $A$ is however allowed to send messages at time instants $(6m+2)\widehat{T}$ and $(6m+3)\widehat{T}$, for $m \geq 0$. Consequently, in Fig. 6(a), although $A$ is ready to send its first message, say *msg1*, at time 0, it can only send it at time $2\widehat{T}$. This message is then received by $B$ at time $3\widehat{T}$. After sending *msg1*, module $A$ sends *msg2* only after a computation delay of 1 cycle. Hence, $A$ is ready to send the second message, *msg2*, at time $6\widehat{T}$. Unfortunately, since Sarmenta *et al*'s approach forbids $A$ from sending messages at time instants ($6m\widehat{T}$), *msg2* actually gets sent at $8\widehat{T}$, and is sampled by $B$ at $9\widehat{T}$. Module $A$ can then send *msg3* at time $12\widehat{T}$. Thus, two messages are sent from $A$ to $B$ in $12\widehat{T}$.

In contrast, our approach does not restrict when the sender can send messages. Instead, the arrival of messages at the receiver's end is appropriately delayed so that every message arrives outside the setup-hold window of receiver clock ticks. In the example of Fig. 6, our approach allows *msg1* to be sent by $A$ at time 0. This message is then sampled by $B$ at time $3\widehat{T}$. After 1 cycle of computation, $A$ can send *msg2* sent at time $4\widehat{T}$, and this is received by $B$ at time $6\widehat{T}$. $A$ can now send *msg3* at time $8\widehat{T}$ after 1 cycle of computation, and this message is received by $B$ at time $9\widehat{T}$. The sequence of communication then repeats with $A$ sending *msg4* at time $12\widehat{T}$. As can be seen from the above example, our approach allows three messages to be sent from $A$ to $B$ in $12\widehat{T}$, whereas only two are sent in the same time by Sarmenta *et al*'s scheme. This shows the performance advantages obtainable using our method.

Sarmenta *et al*'s scheme can be modified slightly to overcome the performance degradation demonstrated above. This can be done by designing a *wrapper* around the sender module, and by feeding the *send-inhibit* signal from the interface to the wrapper instead of sending it to the sender module. Thus, the sender is never inhibited from sending; instead the wrapper accepts messages from the sender and buffers them if needed, presenting messages to Sarmenta *et al*'s interface circuit only when the *send-inhibit* signal is not asserted. Note, however, that the use of buffers in the wrapper adds to the hardware.

## 6 Case studies

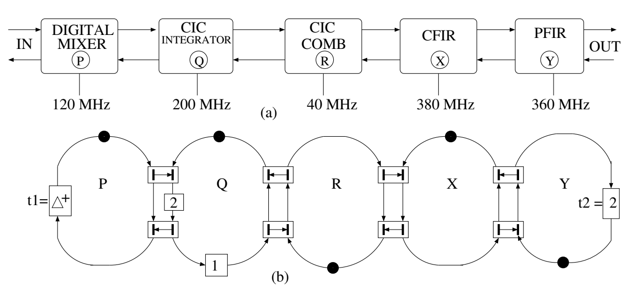


Figure 7. Digital Down conversion (DDC)

In this section we show how analyzing communication protocols simplifies the total interfacing requirements in three different SoC examples. The protocol details of these SoCs, their $NuSMV$ models and details of their analyses can be found in [7]. Figs. 7(a) and (b) show the block diagram and netchart of a *Digital Down Conversion* (DDC) SoC. Here, the clock periods of modules $P$, $Q$, $R$, $X$ and $Y$ are $285\widehat{T}$, $171\widehat{T}$, $855\widehat{T}$, $90\widehat{T}$ and $95\widehat{T}$ respectively, where $\widehat{T}$ denotes the *gcd* clock period. We assume that $SU = H = 50\widehat{T}$ and $D = 63\widehat{T}$ for modules $P$, $Q$ and $R$, and $SU = H = 10\widehat{T}$ and $D = 40\widehat{T}$ for modules $X$ and $Y$. We also assume negligible $\delta_{CD}$ for all modules. Using the method described in Section 4.3, we then determine all time instants when messages can be sent from different modules. We find that out of a total of eight possible interfaces (denoted as "Tot. Int." in Table 1), three do not require any interface circuits, since *all sender cycles are safe* for transmitting messages in these interfaces. The number of such interfaces is denoted as "Int. not reqd." in Table 1. In four of the remaining five interfaces, the protocol guarantees that messages are sent only in safe cycles, thereby obviating the need for interfaces. We use "S" to denote such interfaces in the column titled "Our approach" in Table 1. Thus, the entry "4:S" in the first row of the table signifies that the DDC example has 4 interfaces that do not require interface circuits because the protocol ensures that messages are sent only in safe cycles. In the DDC example, we therefore need to implement only one interface. Our analysis [7] indicates that this can be implemented using a *2-Delay* solution. This is represented as the entry "1:2D" (one interface with a *2-Delay* solution) in the column titled "Our approach" in Table 1. Note that Sarmenta et al's approach [10] does not make use of the protocol information, but constructs the interface based on worst case communication. Accordingly, in the DDC example, two interfaces using double buffering (or "DB") and three interfaces using simple single buffering (or "SB") need to be implemented [7]. The number of "DB" and "SB" interfaces in Sarmenta et al's solution are listed in the column labeled "Sarmenta et al" in Table 1. In addition to the DDC example, we also studied two MPEG2 SoCs as part of our case studies. The architecture, netcharts, and detailed analyses of these examples are omitted here for lack of space, and can be found in [7]. A comparison of the interface requirements for both these examples using our approach and using Sarmenta et al's approach is presented in Table 1, where we use $M1$ and $M2$ to refer to the two MPEG2 SoC examples. Thus, in the case of $M2$, nine interfaces can be implemented as *1-Delay* solutions, while one interface must be implemented as a *2-Delay* solution. Note that none of the interfaces required more than two distinct delays to be used in the implementation. It is clear that significant simplifications in interface design are obtained by our method in comparison with Sarmenta et al's method. As dis-

| | *Tot. Int.* | *Int. not reqd.* | Our approach | Sarmenta *et al* |
|---|---|---|---|---|
| DDC | 8 | 3 | 4:$S$, 1:2D | 3:SB, 2:DB |
| M1 | 22 | 10 | 11:$S$, 1:1D | 4:SB, 8:DB |
| M2 | 34 | 18 | 6:$S$, 9:1D, 1:2D | 3:SB, 4:DB |

Table 1. Comparison with Sarmenta's method

cussed in Section 5.4, this translates to advantages in terms of hardware, power and overall system performance.

## 7 Conclusion

In this paper, we have investigated the problem of interface design for rationally clocked modules in GALS systems. We presented delay-augmented netcharts as a formalism for specifying communication protocols between various modules, and showed that analysis of these protocols can lead to significant optimizations in the design of synchronization circuits. Our case studies show that protocol information and knowledge of clock ratios often obviate the need for interface circuits. We believe that this has important implications for the choice of flow-control protocols used in future SoCs. Future work must investigate techniques to explore the combined space of flow-control protocols and synchronization mechanisms, while minimizing objective functions that combine hardware overhead, performance penalties and vulnerabilities to variations in delays.

## References


[1] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *Proc. of ICCAD'99*, pages 309–315, Nov. 1999.

[2] A. Chakraborty and M. R. Greenstreet. Efficient self-timed interfaces for crossing clock domains. In *Proc. of ASYNC'03*, pages 78–88, May 2003.

[3] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. of CAV 2002*, volume 2404 of *LNCS*. Springer, July 2002.

[4] R. Dobkin, R. Ginosar, and C. P. Sotiriou. Data synchronization issues in GALS SoCs. In *Proc. of ASYNC'04*, pages 170–179, 2004.

[5] M. R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication.* PhD thesis, Princeton University, 1993.

[6] J. Mekie, S. Chakraborty, and D. K. Sharma. Evaluation of pausible clocking for interfacing high-speed IP cores in GALS systems. In *Proc. of VLSI'04*, pages 559–564, January 2004.

[7] J. Mekie, S. Chakraborty, G. Venkataramani, P. S. Thiagarajan, and D. K. Sharma. Interface design for rationally clocked GALS systems. Technical Report TR-06-18, CFDVS, IIT Bombay, 2006.

[8] M. Mukund, K. N. Kumar, and P. S. Thiagarajan. Netcharts: Bridging the gap between hmscs and executable specifications. In *Proc. CONCUR '03, Springer LNCS 2761*, pages 296–310, 2003.

[9] A. Roychoudhury, P. S. Thiagarajan, T.-A. Tran, and V. Zvereva. Automatic generation of protocol converters from scenario-based specifications. In *Proc. of RTSS'04*, pages 447 – 458, 2004.

[10] L. F. G. Sarmenta. Synchronous communication techniques for rationally clocked systems. Master's thesis, Massachusetts Institute of Technology, May 1995.

[11] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[12] K. Y. Yun and A. E. Dooply. Pausible clocking based heterogeneous systems. *IEEE Transactions on VLSI systems*, 7(4):482 – 487, December 1996.