Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks

MohammadReza Mousavi[1], Paul Le Guernic[2], Jean-Pierre Talpin[2],
Sandeep Kumar Shukla[3], Twan Basten[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands

[2]INRIA/IRISA, Rennes, France [3]Virginia Tech., Blacksburg, USA

Abstract

We lay a foundation for modeling and validation of asynchronous designs in a multi-clock synchronous programming model. This allows us to study properties of globally asynchronous systems using synchronous simulation and model-checking toolkits. Our approach can be summarized as automatic transformation of a design consisting of two asynchronously composed synchronous components into a fully synchronous multi-clock model preserving behavioral equivalence. The ultimate goal of this research is to provide the ability to model and build GALS systems in a fully synchronous design framework and deploy it on an asynchronous network preserving all properties of the system proven in the synchronous framework.

1. Introduction

Synchronous languages have been extensively used in the (co-)design of software and hardware systems [4]. Applying synchronous languages to real-world designs revealed their strong and weak points over time. Abstract and easy to learn and use syntax, formal and succinct semantics (which paved the way for efficient simulation and verification tools) are among the strong points of such languages. However, the synchronous assumption turns out to be a limiting factor. On one side of the spectrum, in distributed real-time systems, providing a single, fully synchronized clock over distributed nodes may be very expensive or actually infeasible. On the other side of the spectrum, in nano-scale system design, the propagation delay of the clock over the chip size becomes a major obstacle in providing a single synchronized clock. Thus, all these domains call for a mix of synchronous and asynchronous design patterns.

Globally Asynchronous, Locally Synchronous (GALS) designs have emerged in the recent years in response to the above mentioned challenges and have received major attention from the system level design community. In GALS design, the system is composed of synchronous components that have their local synchronous clock structures and communicate using asynchronous schemes. There have been several attempts to formalize GALS design (see for example, [14, 3]).

Problem Statement. In this paper, we address the problem of modeling and validating asynchronous composition of synchronous components in the multi-clock synchronous programming framework SIGNAL. The main goal of such an approach is to leverage the simulation and model checking toolkit existing for such frameworks [1]. Our solution can be seen as a formal methodology for composing existing IP blocks, designed with synchronous assumptions, in an asynchronous fashion to satisfy the demands of tomorrow's GALS designs.

Our approach can be summarized as developing a design consisting of asynchronously composed components within a synchronous framework. Since true asynchrony does not exist in synchronous design frameworks, we seek for a desynchronizing protocol to match the asynchronous model. Finding such a protocol brings about the possibility of formally investigating the behavior of synchronous components in asynchronous environment.

In Section 2, we present some related work. Section 3 contains definitions of the SIGNAL language. Subsequently, we define an ideal desynchronizing protocol using unbounded FIFO channels and prove it correct in Section 4. Since an unbounded FIFO channel cannot be implemented in SIGNAL, this protocol is only an ideal model but it can be used as a reference model for other non-perfect desynchronization protocols. Then, we provide the conditions under which this design can be refined to a network of bounded and thus implementable FIFO channels. In Section 5, we propose an implementation by defining the FIFO channels present in a design to estimate the appropriate FIFO channel size in practice. Finally, Section 6 summarizes the results and presents the concluding remarks.

2. Related Work

In [2], distributing SIGNAL programs is studied under synchronous conditions. Since all components are assumed to work with a single master clock, there, the size of the buffer is naturally restricted to one. In [16], the problem of decomposing SIGNAL programs into components is studied and a handshaking mechanism is proposed for the asynchronous communication among components. Our work extends the results of [2, 16] to asynchronous settings where handshaking is removed or reduced to some extent.

In [14], the issue of communication-based design is addressed. It introduces the idea of *Behavior Adapters* in order to interface two (possibly mismatching) input and output signals. FIFO queues are then proposed as primitive communication channels. It is claimed there that unbounded FIFO channels are ideal communication mechanisms for asynchronous designs (expressed as Abstract Co-

Design Finite State Machines or ACFSMs for short). Then ACFSMs are refined into ECFSMs (for Extended CFSMs) which contain a network of bounded FIFO channels and a blocking mechanism or a lossy channel to overcome the rate mismatch problem.

Our contribution to the work of [14] can be summarized in the following two issues: First, we formalize the concepts of asynchronous design in the SIGNAL model. This formalization provides us the possibility to prove the claim of ideal asynchrony with unbounded buffers and the conditions for refining it to bounded buffers. Secondly, we propose a practical design to estimate the size of buffers in the refined design so that we can decrease the amount of blocking for normal system behavior.

The work of Berry and Sentovich in [5] studies the issue of asynchronous interaction between synchronous Esterel programs. There, the authors solve the problem of overwriting messages due to asynchrony by blocking the sender when the single-place buffer is full. Although in this way the buffer size is restricted to 1, the parallelism and pipelining is decreased.

In [7], distribution of synchronous sequential programs (in the style of Esterel programs) is discussed. In this approach, the asynchronous interaction between the components is encapsulated in *send* and *receive* commands and the main effort is invested in exploring the appropriate places for send and receives in order to minimize communication and maximize parallelism. However, the issue of buffer sizes is left implicit and remains to be addressed by investigating the semantics of (asynchronous) send and receive commands which in one way or the other involves this issue.

Implementing asynchronous systems using synchronous languages is also studied in [8]. After defining a generic semantic model for synchronous and asynchronous computation, the attention is focused on implementing communication mechanisms such as mutual exclusion mechanisms and rendez-vous. Although these mechanisms can be very handy in asynchronous system design, the paper does not suggest any process starting from synchronous component designs and arriving in components instrumented with these structures. This is one of the goals of the present paper (with respect to FIFO channels).

The work of [9] models asynchrony (interleaving semantics) in the I/O Automata model using a synchronous communication mechanism. However, due to differences between the models of computation (such as the input enabling assumption in I/O Automata), the notion of buffer is internalized inside the semantics of [9] and is not addressed explicitly.

3. System Design in SIGNAL

The abstract syntax of core SIGNAL is given in Figure 1. In this syntax, a SIGNAL program is decomposed into several components. Components are assumed to work synchronously and in parallel. Decomposition of a SIGNAL program can be the result of reusing a number of COTS (Commercial Off The Shelf) components or decomposition techniques based on graph partitioning (see for example [12, 16]). A single component consists of a number of signal definitions (we omit component names when it causes no confusion). The set of all signal names is denoted by X with typical members $x, y, z, \ldots$. There are a few primitive operators in SIGNAL allowing for definition of basic processes. The expression $x = \texttt{pre}\ val\ y$ defines that the signal named x holds the previous value of signal y, and it is initially set to val (thus, x and y are synchronous). The equation $x = y\ \texttt{when}\ z$ defines x to have the value of y when z is present and *true*; otherwise x is undefined. The equation $x = y\ \texttt{default}\ z$ defines x by y when it is present and otherwise by z.

$$
\begin{array}{lcll}
Program & ::= & PName = & Component \mid \\
 & & & Component \mid\mid_s Program \\
Component & ::= & CName : Expressions & \\
Expressions & ::= & Expression \mid Expression, Expressions & \\
Expression & ::= & x = \texttt{pre}\ val\ y \mid & \\
 & & x = y\ \texttt{when}\ z \mid & \\
 & & x = y\ \texttt{default}\ z \mid & \\
 & & x = f(y, z, \ldots) &
\end{array}
$$

Figure 1. Abstract Syntax of SIGNAL

Apart from primitive operators, we assume existence of a number of arithmetic operators represented by f (and in particular, equality, denoted by $==$) that performs a computation on synchronously available arguments. In our examples of SIGNAL programs, we use $\hat{}x$ as a shorthand for $true\ \texttt{when}\ (x == x)$ meaning "the clock of x". We use nested definition of signals as a shorthand for a number of signal definitions.

Example 1 (One-Place Buffer) To specify a single place buffer, first we give a specification of a single cell memory, as follows:

$$
\begin{array}{lcl}
data & = & msgIn\ \texttt{default}\ (\texttt{pre}\ 0\ data) \\
msgOut & = & data\ \texttt{when}\ \hat{}msgOut
\end{array}
$$

The above program allows for independent read and write accesses (denoted by *msgIn* and *msgOut*) and the memory cell keeps the last written value and is initialized to 0. The independence between the rate of input and output signals shows the essence of polychrony in SIGNAL design. This provides us the possibility for desynchronizing designs. To change this initial specification to a single place buffer where causality between reads and writes are forced and the first in first out principle is observed, we have to make the following changes to the program:

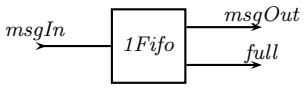
$$
\begin{array}{ll}
data & = (msgIn\ \texttt{when}\ (\texttt{not}\ full))\ \texttt{default} \\
 & \quad (\texttt{pre}\ 0\ data) \\
msgOut & = data\ \texttt{when}\ (\hat{}msgIn\ \texttt{default}\ full) \\
in & = \hat{}msgIn\ \texttt{default}\ false \\
out & = \hat{}msgOut\ \texttt{default}\ false \\
full & = ((\texttt{pre}\ in\ false) \wedge not(\texttt{pre}\ out\ false))\ \texttt{default} \\
 & \quad (\texttt{pre}\ full\ false) \\
\hat{}data & = (\hat{}msgIn\ \texttt{default}\ \hat{}msgOut)\ \texttt{default}\ \hat{}full \\
\hat{}full & = \hat{}in = \hat{}out
\end{array}
$$

Schematic view of the one-place buffer, specified above, together with a sample behavior of this program is depicted in Figure 2. This program only writes the value of input into the buffer if the buffer is not full, and only allows to take the data from the output, if the buffer contains some

$$
\begin{array}{lcl}
[[x = \texttt{pre } val\ y]] & = & \{b_{|\{x,y\}} | tags(b(x)) = tags(b(y)), b(x)(t(b(y)_1)) = val, \forall i \in I\!N, b(x)(t(b(y)_{i+1})) = b(y)(t(b(y)_i))\} \\
[[x = y \texttt{ when } z]] & = & \{b_{|\{x,y,z\}} | tags(b(x)) = tags(b(y)) \cap \{t | t \in tags(b(z)) \wedge b(z)(t) = true\}, \forall t \in tags(b(x)), b(x)(t) = b(y)(t)\} \\
[[x = y \texttt{ default } z]] & = & \{b_{|\{x,y,z\}} | tags(b(x)) = tags(b(y)) \cup tags(b(z)), \\
 & & \forall t \in tags(b(x)), (b(x)(t) = b(y)(t) \wedge t \in tags(b)(y)) \vee (b(x)(t) = b(z)(t) \wedge t \notin tags(b(y)\ \wedge\ t \in tags(b)(z)))\}
\end{array}
$$

Table 1. Semantics of elementary SIGNAL equations

data. The buffer is initially set to be empty and becomes full when a data item is written to it provided that it is not taken out at the same moment.



msgIn		1	2			3
in	*false*	*true*	*true*	*false*	*false*	*true*
full	*false*	*false*	*true*	*true*	*false*	*false*
data	0	1	1	1	1	3
out	*false*	*false*	*false*	*true*	*false*	*true*
msgOut				1		3

Figure 2. Sample behavior of a 1-place buffer

In this paper, we use the tagged model [11] of the SIGNAL language (see [10] for a detailed explanation of SIGNAL and its semantics). The agged model of [10] defines behavior of a SIGNAL process (program) in terms of independent signals that may have different time scales. Thus, in this *polychronous* model, time is not necessarily linear. Denotation of a SIGNAL process is defined as follows.

Definition 1 (Denotation of a Process) Time is taken from a set T (called *tags*) with a partial order $\preceq$. This notion of time allows for specification of distributed processes with local clocks (possibly with different rates) and synchronization points. Events are values of a signal at points of time (taken from a given set V, here we use integers and booleans as possible values for events). The set of events $\varepsilon = T \times V$ is a relation between tags and values (V). We denote the time of event e by $t(e)$. A signal $s \in S : T \rightarrow V$ is a partial function defining the value of a signal over a discrete chain of tags , denoted by $tags(s)$ (thus, events in a signal are internally ordered and their tags are assumed to be well-founded). For a signal s, we denote its i'th event ($i \in I\!N$) by s_i and the sub-chain of length $n+1$ starting from i'th event by $s_{i...i+n}$. The set of events of signal s up to the point t is denoted by $[s]_t$ and the length of the chain of signal s is denoted by $|s| \in I\!N \cup \{\infty\}$. In this paper, we are concerned with processes containing infinite (reactive) behavior.

A behavior $b \in B : X \rightarrow S$ is a partial function defining signal values for different signal names from the set X. The domain of a behavior b is denoted by $vars(b)$ and represents signal names taking part in this behavior. A process $P \subseteq B$ is a set of behaviors over a common set of signal names defining different possible behaviors of a program (a component). Two processes P and Q are equal ($P = Q$) if they contain the same set of behaviors. Projection of a behavior b on a set of variables $var \subseteq X$ (denoted by $b_{|var}$) is defined by restricting the domain of the behavior to var. Projection of a process P on var (denoted by denoted by $P_{|var}$) is defined by projection of all its member behaviors. Its dual, denoted by $b_{\backslash var}$ (similarly, $P_{\backslash var}$), is a short-hand for $b_{|vars(b)\backslash var}$.

Semantics of basic equations in SIGNAL is defined in Table 1, in terms of the denotation of the basic processes.

Definition 2 (Stretching and Stretch-Equivalence) A behavior c is a stretching of behavior b denoted by $b \leq c$ if and only if $vars(b) = vars(c)$ and there exists a bijection $f : T \rightarrow T$ such that

1. $\forall t, u \in T, t \preceq u \Leftrightarrow f(t) \preceq f(u)$
2. $\forall t \in T, t \preceq f(t)$
3. $\forall x \in vars(b), tags(c(x)) = f(tags(b(x)))$
4. $\forall x \in vars(b), \forall t \in T, b(x)(t) = c(x)(f(t))$

Intuitively, stretching changes the time scale of behaviors while preserving the causal orderings and event synchronizations. Two behaviors b and c are stretch equivalent, denoted by $b \lessgtr c$, if and only if there exists a behavior d such that $d \leq b$ and $d \leq c$. The definition of stretching and stretch equivalence is extended to processes using element-wise comparison of member behaviors. Stretch closure of a process P is denoted by P^* and is defined as $\{b | \exists c \in P, b \lessgtr c\}$. A process P is stretch-closed if and only if $P^* = P$.

Definition 3 (Synchronous Parallel Composition) Semantics of synchronous parallel composition (denoted by $||_s$) is defined as follows:

$$P\ ||_s\ Q = \{d_{|X \cup Y} | \exists (b, c) \in P \times Q, d_{|X} = b \wedge d_{|Y} = c\}$$

where $X = vars(P)$ and $Y = vars(Q)$.

Lemma 1 All SIGNAL programs are stretch-closed.

Proofs of lemmas and theorems are omitted for brevity. See [13] for proofs.

Definition 4 (Relaxation and Flow Equivalence) Behavior c is a relaxation of b, denoted by $b \sqsubseteq c$ if and only if $vars(b) = vars(c)$ and for all $x \in vars(b), b_{|\{x\}} \leq c_{|\{x\}}$. Intuitively, relaxation stretches different signals with possibly different rates (which may not preserve causal ordering and event synchronizations). Two behaviors b and c are flow equivalent, denoted by $b \approx c$, if and only if $\exists d, b \sqsubseteq d \wedge c \sqsubseteq d$. Relaxation of processes is defined similarly by an element-wise comparison of behaviors.

Definition 5 (Renaming Signals) Behavior $b[y/x]$ (similarly, process $P[y/x]$) is the result of renaming signal name x by the fresh signal name y ($y \notin vars(P)$) in b (similarly, in all behaviors in P).

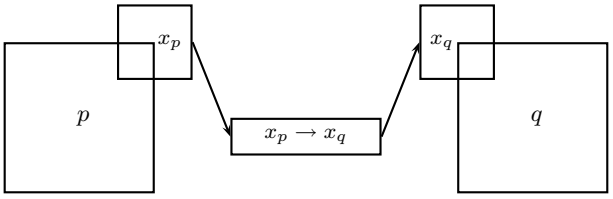


Figure 3. Desynchronization: Schematic View

4. Desynchronization

The aim of this section is to give an implementation (a concrete way of modeling) of asynchronous parallel composition in SIGNAL using synchronous (polychronous) constructs. We start with defining the notion of asynchronous parallel composition and restrict it to a causally ordered distributed setting. Then, we show that replacing explicit data dependencies of two components with an unbounded FIFO buffer is a correct implementation of asynchronous causal parallel composition. Then, we set out to replace the unbounded FIFO with FIFO channels of a bounded size. To do this, we investigate the conditions under which this leads to a correct implementation.

4.1 Desynchronizing with Unbounded FIFOs

The main idea behind desynchronization is implementing asynchronous parallel composition in SIGNAL designs using FIFO channels. Figure 3 depicts a schematic view of desynchronization. In this process, we introduce a FIFO channel for each data dependency instead of using a shared variable for this data dependency and thus we relax the synchrony between the two components. To prove correctness of this approach, we start with proving the fact that if this replacement is done by an unbounded FIFO channels, it is indeed a correct implementation of an asynchronous-causal parallel composition:

Definition 6 (Asynchronous Parallel Composition) Asynchronous parallel composition of two processes P and Q is defined as follows [10]:

$$\begin{aligned} P \mid\mid_a Q = \{ & d_{|X\cup Y} | \exists (b,c) \in P \times Q, \\ & d_{\backslash Y} \lesseqgtr b_{\backslash Y} \wedge d_{\backslash X} \lesseqgtr c_{\backslash X} \wedge \\ & b_{|X\cap Y} \sqsubseteq d_{|X\cap Y} \wedge c_{|X\cap Y} \sqsubseteq d_{|X\cap Y} \} \end{aligned}$$

where $X = vars(P)$ and $Y = vars(Q)$. This definition defines that when two processes are put in asynchronous parallel composition, their internal actions may be stretched (due to different relative time scales of their local platforms) and their common variables (their communication media) may be stretched with different rate (denoted by relaxation notation, due to different characteristics of communication channels).

Corollary 1 For two stretch closed processes P and Q such that $vars(P) \cap vars(Q) = \emptyset$, it holds that $P \mid\mid_s Q = P \mid\mid_a Q$.

Definition 7 (Asynchronous Causal Parallel Composition) If two processes P and Q share a variable x ($x \in vars(P) \cap vars(Q)$), then there is an *explicit data-dependency* between P and Q. A causal ordering between P and Q (or vice-versa) on data-dependency x is denoted by $P \leq_x Q$ ($Q \leq_x P$, respectively). This means that P is the producer of x and Q is its consumer (x appears only in the right-hand-side of the assignments in P and in the left-hand-side in Q). Note that in a general SIGNAL program, for a data dependency x, it is not necessarily true that $P \leq_x Q$ or $Q \leq_x P$. However, the above restriction is not a strong constraint for most executable applications.

We define asynchronous causal parallel composition of two processes (denoted by $||_{\leq,a}$) as follows:

$$\begin{aligned} P \mid\mid_{\leq,a} Q = \{ & d_{|X\cup Y} | \exists (b,c) \in P \times Q, \\ & d_{\backslash Y} \lesseqgtr b_{\backslash Y} \wedge d_{\backslash X} \lesseqgtr c_{\backslash X} \\ & \wedge b_{|X\cap Y} \sqsubseteq d_{|X\cap Y} \wedge c_{|X\cap Y} \sqsubseteq d_{|X\cap Y} \\ & \wedge \forall x \in X \cap Y, P \leq_x Q \Rightarrow b_{|\{x\}} \leq c_{|\{x\}} \\ & \wedge \forall y \in X \cap Y, Q \leq_y P \Rightarrow c_{|\{y\}} \leq b_{|\{y\}} \} \end{aligned}$$

The above definition, as asynchronous parallel composition, allows for asynchronous communication and stretching internal behavior of processes. Furthermore, it asserts that if P depends on Q for x, it cannot read x before it is written by Q, and vice versa.

Corollary 2 For two processes P and Q such that $vars(P) \cap vars(Q) = \emptyset$, it holds that $P \mid\mid_a Q = P \mid\mid_{\leq,a} Q$.

Definition 8 (Asynchronous FIFO Channel) An (unbounded) asynchronous FIFO channel $AFifo_{x\rightarrow y}$ with input port x and output port y is the smallest stretch-closed process P satisfying $vars(P) = \{x, y\}$ and $P_{|\{x\}} \leq P_{|\{y\}}[x/y]$.

Theorem 1 Suppose that P and Q are stretch-closed processes and x is a shared signal produced by P and consumed by Q ($P \leq_x Q$), then

$$\begin{aligned} & (P \mid\mid_{\leq,a} Q)_{\backslash\{x\}} = \\ & \quad ((P[x_P/x] \mid\mid_{\leq,a} Q[x_Q/x]) \mid\mid_s AFifo_{x_P \rightarrow x_Q})_{\backslash\{x_P, x_Q\}} \end{aligned}$$

Due to Theorem 1, if we continue the process of desynchronization, we get a network of FIFO channels (named R) such that: $(P \mid\mid_{\leq,a} Q)_{\backslash I} = ((P' \mid\mid_{\leq,a} Q') \mid\mid_s R)_{\backslash I'}$, where $I = vars(P) \cap vars(Q)$, P' and Q' are results of iterative replacements of explicit data dependencies with fresh variables and I' is the set of such fresh variables. Since all explicit data dependencies between P and Q are resolved ($vars(P') \cap vars(Q') = \emptyset$), with the help of Corollaries 1 and 2, we achieve complete desynchronization as follows:

$$\begin{aligned} (P \mid\mid_{\leq,a} Q)_{\backslash I} &= ((P' \mid\mid_{\leq,a} Q') \mid\mid_s R)_{\backslash I'} \\ &= (P' \mid\mid_s Q' \mid\mid_s R)_{\backslash I'} \end{aligned}$$

However, an unbounded FIFO channel is only a semantical object and does not have a corresponding SIGNAL component, neither is it implementable in embedded system designs. Thus, we would like to replace the unbounded channel with a bounded one. This is certainly not possible in all designs, however, we investigate this possibility in the next section.

4.2 Desynchronizing Using Bounded FIFOs

To restrict the desynchronizing protocol to a network of bounded FIFOs, we first specify the semantics of a bounded network. Then, we give a semantic characterization of processes that can show the asynchronous behavior if they are composed using bounded FIFOs.

Definition 9 (Bounded N-FIFO) A bounded n-FIFO, denoted by $nFifo_{x\to y}$ is the largest process P with $vars(P) = \{x, y\}$, satisfying the following condition:

$$P \subseteq AFifo_{x\to y} \wedge \forall b \in P, \forall t \in T, |[b(x)]_t| \leq n + |[b(y)]_t|$$

The above definition specifies that a bounded n-FIFO should, first, satisfy the FIFO characteristics and, second, at each point of time the number of writes can deviate from the number of reads so far by at most n. Next, we give the characterization of processes that share only a single variable and for which this explicit data dependency can be replaced by a bounded FIFO buffer.

Lemma 2 If $vars(P) \cap vars(Q) = \{x\}$ then $(P \mid\mid_{\leq,a} Q)_{\setminus\{x\}} = (P[x_P/x] \mid\mid_s Q[x_Q/x] \mid\mid_s nFifo_{x_P\to x_Q})_{\setminus\{x_P,x_Q\}}$, if and only if:

1. $P \leq_x Q \wedge$
2. $\forall (a,b) \in P \times Q, a_{|\{x\}} \leq b_{|\{x\}} \Rightarrow$
   $\exists (a',b') \in P \times Q, a'_{\setminus\{x\}} = a_{\setminus\{x\}} \wedge b'_{\setminus\{x\}} = b_{\setminus\{x\}} \wedge$
   $\forall i \in I\!N, t(b'(x)_i) \preceq t(a'(x)_{i+n})$

The above lemma defines necessary and sufficient conditions for two components P and Q so that if they are connected by an $nFifo$, they can perform the same behavior as when they are put in an arbitrary asynchronous network. That is, all read actions of component Q from x can be delayed at most for n more write actions of P on the same variable (thus, preventing buffer overflow).

Next, we generalize Lemma 2 to a network of FIFO channels in both directions:

Theorem 2 Consider the two processes P and Q, let I and O be the subsets (partitions) of $vars(P) \cap vars(Q)$ such that $\forall x \in I, Q \leq_x P$ and $\forall y \in O, P \leq_y Q$. If
$\forall (a,b) \in P \times Q,$
$(\forall x \in I, a_{|\{x\}} \leq b_{|\{x\}} \wedge \forall y \in O, a_{|\{y\}} \leq b_{|\{y\}}) \Rightarrow$
$\exists (a',b') \in P \times Q, a_{\setminus(I\cup O)} = a'_{\setminus(I\cup O)} \wedge b_{\setminus(I\cup O)} = b'_{\setminus(I\cup O)} \wedge$
$\forall i \in I\!N, \forall x \in I, t(b'(x)_i) \preceq t(a'(x)_{i+n_x}) \wedge$
$\forall y \in O, t(a'(y)_i) \preceq t(b'(y)_{i+n_y})$
then $(P \mid\mid_{\leq,a} Q)_{\setminus(I\cup O)} = (P' \mid\mid_s Q' \mid\mid_s R)_{\setminus(I'\cup O')}$, where P' and Q' are the result of replacing all variables $x \in I \cup O$ with fresh variables x_P and x_Q, respectively, I' and O' are the sets of such fresh variables and R is the network of $n_x Fifo_{x_P\to y_Q}$ and $n_y Fifo_{y_Q\to y_P}$ channels.

Note that the proposed approach and in particular the above theorem holds for channels with single-producer and single-consumer components. In other words, it is assumed that a shared variable can only be shared by two components. This is not a very restrictive assumption since for multiple-producer, multiple-consumer shared variables, one can make use of standard copy (fork) and merge (join) components to copy the shared channel for several components and join several write attempts of different components into one channel.

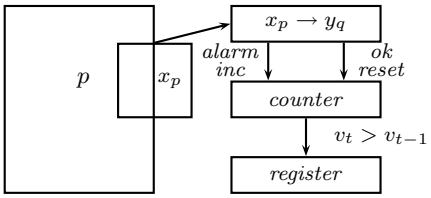


Figure 4. Instrumented FIFO

5. Approximating Buffer Sizes

In this section, we implement the desynchronization ideas inside the SIGNAL model. To do this, first, we have to define the implementation of the network of FIFO processes. Although the characterization we gave in the previous section can be formally checked on the semantics of components, it does not give a constructive way of determining the buffer size. Thus, we propose a practical approach in this section, with which we are able to estimate the buffer size (for a given environment).

5.1 Implementing FIFO Channels

An *nFifo* channel can be implemented using a composition of n *1Fifo*'s defined in Example 1, as follows:

$$\begin{array}{ll} nFifo_{x_0\to x_n} & = 1Fifo_{x_0\to x_1}[full_1, in_1, out_1/full, in, out] \\ & \mid\mid_s \ldots \\ & \mid\mid_s 1Fifo_{x_{n-1}\to x_n}[full_n, in_n, out_n/full, in, out] \\ in_i & = (in_{i-1} \texttt{ when } not(full_i)) \texttt{ default } in_i \\ out_i & = (out_{i+1} \texttt{ when } full_i) \texttt{ default } out_i \ (0 < i < n) \\ alarm & = (full_1 \wedge \ldots \wedge full_n) \texttt{ when } x_P \\ ok & = \texttt{not } (alarm) \end{array}$$

Note that the *alarm* signals is synchronized with *write* attempts. Thus, for an unsuccessful attempt to write to the buffer the *alarm* signal is raised. Its negation describes a successful write attempt.

5.2 Instrumenting FIFO Channels

Using the implementation of an *nFifo* channel of the previous subsection, we are able to design the circuitry around the FIFO channels as shown in Figure 4. In this figure, every time a write signal is received by the channel, if the channel is full and the data cannot be inserted to the channel, an *alarm* signal is raised by the channel which in turn results in an *inc* event of the corresponding buffer. An *ok* signal from the FIFO results in resetting the counter. We keep the maximum value of the counter in a register which represents the number of times we consecutively missed a write to the buffer (for sake of brevity, we do not give detailed SIGNAL implementations of the counter and register components here). Designers can start with a set of behaviors and a rough guess of the needed buffer size and use the

instrumented FIFO network (replacing explicit data dependencies) to find the right estimation of the buffer size. This is done by simulating the behavior of the design for a given environment, observing the values in the counters, incrementing the buffer size by these values, and iterating the simulation till no alarm is raised. This process guarantees that for a set of (normal) behaviors, no buffer overflow will happen. However, since the designer does not necessarily feed all possible behaviors into the design, we need a feedback loop to prevent loosing data in exceptional cases of buffer overflow.

To do this, we can use the conjunction of all $full_i$ signals to mask the clock of the producer component. Masking the clock of the producer may be too naive for some critical designs. In such cases, different service levels should be implemented in which the rate of production and consumption of data items can be tuned. The necessity to change the service level can then be indicated by observing the status of communication between components using the FIFO buffers between them. For a survey of such techniques in practical GALS designs see [15].

Verification of the desynchronized design consists of checking that no *alarm* signal is raised. In case of failing to prove this, the error trace may help us finding the input sequence resulting in *alarm*. This input can be added to our simulation data. Then, we can re-iterate the process by simulating with the new test-data, estimating the sufficient buffer size and coming back to the verification phase. If for a set of input signals no alarm is raised (registers of instrumented FIFOs all show zero) then the design is correct for those inputs (that is according to Theorem 2) the proposed FIFO network faithfully implements asynchronous communication.

6. Conclusion

In this paper, we established the theoretical model of asynchronous composition in SIGNAL and its implementation using FIFO buffers. In addition to that, we proposed a practical design template to estimate the buffer size. The proposed approach allows for efficient analysis and implementation of asynchronous designs. Furthermore, it brings about the possibility of specifying GALS systems in the synchronous framework and benefitting from the tooling around it.

Studying compositionality and stability issues in the buffer size proof and estimation remains as one of our future research topics. We are also looking at constructive algorithms based on the clock dependency graph to make the buffer size estimation and proof automatic. Using a program morphism approach (similar to the approach taken in [6]) is another possibility for simulating the program behavior and estimating the buffer size.

7. REFERENCES

[1] The polychrony toolset. `http://www.irisa.fr/espresso/Polychrony`.

[2] P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of SIGNAL programs. In *Proceedings of HICSS-29*, pages 656–665. IEEE Computer Society, 1996.

[3] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J. C. M. Baeten and S. Mauw, editors, *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 162–177. Springer, 1999.

[4] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proceedings of EMSOFT'02*, volume 2491 of *LNCS*, pages 252–266. Springer, 2002.

[5] G. Berry and E. M. Sentovich. An implementation of constructive synchronous programs in POLIS. *Formal Methods in System Design*, 17(2):135–161, 2000.

[6] A. Gamatié, T. Gautier, and L. Besnard. Modeling of avionics applications and performance evaluation techniques using the synchronous language signal. To Appear in *Proceedings of SLAP'03*, volume 88 of *ENTCS*. Elsevier, 2003.

[7] A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proceedings of EMSOFT'02*, volume 2491 of *LNCS*, pages 266–281. Springer, 2002.

[8] N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proceedings of EMSOFT'02*, volume 2491 of *LNCS*, pages 240–251. Springer, 2002.

[9] R. Kurshan, M. Merritt, A. Orda, S. Sachs. Modelling Asynchrony with a Synchronous Model. *Formal Methods in System Design*, 15(3): 175–199, 1999.

[10] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12(3): 261-304, Apr. 2003.

[11] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12): 1217-1229, Dec. 1998.

[12] O. Maffeïs and P. Le Guernic. Distributed implementation of Signal: Scheduling & graph clustering. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Proceedings of FTRTFT'94*, volume 863 of *LNCS*, pages 547–566. Springer, 1994.

[13] M. Mousavi, P. Le Guernic, J.-P. Talpin, S.K. Shukla, T. Basten. Modeling and Validation of Globally Asynchronous Design in Synchronous Framework. *Technical Report RR-4935*, INRIA, Rennes, France, 2003.

[14] A. Sangiovanni-Vincentelli, M. Sgroi, and L. Lavagno. Formal models for communication-based design. In C. Palamidessi, editor, *Proceedings of CONCUR'00*, volume 1877 of *LNCS*, pages 29–47. Springer, 2000.

[15] M. Singh and M. Theobald. Generalized Latency-Insensitive Systems for GALS Architectures. To appear in: *Proceedings of FMGALS'03*, 2003

[16] K. Wolinski and M. Belhadj. High level synthesis of globally asynchronous locally synchronous circuits. In *Proceedings NATW'94*, 1994.