# Isochronous Networks by Construction

Yu Bai and Klaus Schneider
Embedded Systems Group, University of Kaiserslautern, Germany

***Abstract*—While synchronous system models have many advantages over asynchronous models concerning verification and validation, many implementation platforms do not provide efficient means for synchronization. For this reason, we consider a design flow that starts with a synchronous system model that is then transformed into an asynchronous one for synthesis. In essence, it partitions the synchronous system into a set of asynchronous components that communicate with each other via FIFO buffers. Of course, the synthesized system still has to behave as the original synchronous model, i.e., for each variable exactly the same flow of data values must be observed and only the membership to synchronous reaction steps is no longer explicitly given. In this paper, we prove that this correctness guarantee is given provided that (1) each component knows which of the input values have to be used for the next reaction (endochrony), (2) each component is able to perform the reaction (constructiveness), and (3) components agree on the clocks of their shared variables (isochrony/clock-consistency).**

## I. Introduction

### A. Motivation

Synchronous models offer many advantages to a model-based design like simplified use of formal methods for verification and analysis, deterministic concurrency for replaying once observed simulations, simplified worst-case execution time analysis, and synthesis methods for hardware/software co-design. However, implementing synchrony is often not efficient for large systems: In circuit design, clock propagation became more and more difficult since the clock tree requires a large part of the chip size and of the power consumption. In embedded systems design, the advent of multicore processors is driving the use of multithreaded systems whose communication is typically done via shared memory where synchronization is again expensive.

To benefit from both synchronous models and asynchronous implementations, we develop a design flow that starts with a synchronous model that, after simulation and verification, is partitioned into asynchronous components. However, not every partition is a legal one in the sense that the behavior of the synchronous system will be preserved. We prove in this paper that there are three important properties that have to be fulfilled that we first discuss informally in the following. To this end, we consider system descriptions that are given by guarded actions $\langle \gamma \Rightarrow \alpha \rangle$ where the action $\alpha$ is performed as soon as the guard condition $\gamma$ holds. To describe the behavior of systems, we consider particular streams of values for their input, local and output variables, and the special value $\boxdot$ is used in the synchronous models to denote that at that point of time, the variable does not have a value.

Since these 'values' $\boxdot$ do not appear in the asynchronous implementations, a first requirement is that each component has to know which input values are required to perform its part of a corresponding synchronous reaction step. This property is called *endochrony* which intuitively means that *the component can derive a local clock on its own*. This is often done by relying on *sequential functions* [1], [2] where one of the inputs is always read first, and depending on the value read from it, it is known which input to read next and also which input values are not required at all. However, while all sequential functions are endochronous, there are endochronous functions that are not sequential like the Gustave function shown in Figure 1. Note that also its asynchronous implementation knows which of the values found at the input ports $x_1$, $x_2$, and $x_3$ are needed for the next reaction (and that the synchronous version has no reaction for cases that are not listed).

| $x_1$ | $x_2$ | $x_3$ | $y$ |
|---|---|---|---|
| 1 | 0 | $\boxdot$ | 1 |
| $\boxdot$ | 1 | 0 | 1 |
| 0 | $\boxdot$ | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |

Fig. 1: Gustave Function

A second requirement is the *constructiveness* which means that once the component has received the input values for the next reaction, it must be able to *algorithmically compute the outputs without guessing*. This requirement has been discussed in many papers, e.g. [3] and is required also for the synchronous model without considering desynchronization.

Endochrony and constructiveness are however not sufficient to guarantee a correct desynchronization. In addition, components that communicate via a shared variable must agree on the 'clock' of that variable, i.e., they must agree to which of the synchronous reactions the values of that variable refer to. To demonstrate this, consider a simple system with two components $P$ and $Q$ where $x$ is an input of $P$, $y$ an output of $Q$, and $v_1, v_2$ are outputs of $P$ and inputs of $Q$. The behavior of $P$ is given by the guarded actions $P := \{\langle \mathsf{IsEven}(x) \Rightarrow v_1 = x\rangle, \langle \mathsf{IsEven}(x) \Rightarrow v_2 = 2 \cdot x - 2\rangle, \langle \mathsf{IsOdd}(x) \Rightarrow v_1 = x+1\rangle, \langle \mathsf{IsOdd}(x) \Rightarrow \mathsf{next}(v_2) = 2 \cdot x\rangle\}$ and $Q := \{\langle 1 \Rightarrow y = v_1 + v_2\rangle\}$.

| $x$ | 1 | 3 | 5 | 7 | 9 | ... |
|---|---|---|---|---|---|---|
| $v_1$ | 2 | 4 | 6 | 8 | 10 | ... |
| $v_2$ | $\boxdot$ | 2 | 6 | 10 | 14 | ... |
| $y$ | $\boxdot$ | 6 | 12 | 18 | 24 | ... |

| $x$ | 2 | 4 | 6 | 8 | 10 | ... |
|---|---|---|---|---|---|---|
| $v_1$ | 2 | 4 | 6 | 8 | 10 | ... |
| $v_2$ | 2 | 6 | 10 | 14 | 18 | ... |
| $y$ | 4 | 10 | 16 | 22 | 28 | ... |

Fig. 2: Example Demonstrating the Need of Isochrony

Both components $P$ and $Q$ are endochronous and constructive[1], but still their desynchronization is not correct: Figure 2 shows two synchronous behaviors for two input streams of $x$. As can be seen, the same values are sent through the shared variables $v_1$ and $v_2$, but these refer to different points of time of the synchronous model, and this leads to different values in the output stream $y$. If the two components were desynchronized,

[1] $P$ always reacts if a value arrives at $x$, and $Q$ needs one value at each of its input ports $v_1$ and $v_2$ whose arrivals will trigger a reaction.

978-3-9815370-2-4/DATE14/©2014 EDAA

i.e., we only communicate the values except for the 'value' $\boxdot$, there is no chance for component $Q$ to distinguish between the two cases.

For this reason, *isochrony* is a third requirement to guarantee correct desynchronization. Intuitively, the original definition of isochrony [4] means that two components agree on the clocks of their shared variables. However, its formal definitions were stated differently (based on comparing flows) in the literature, see e.g. [5], [6]. To distinguish between the two versions, we call the original version of isochrony *clock-consistency*, and prove the result of [5], [6] with clock-consistency instead of isochrony.

### B. Related Work

We share the same spirit with the desynchronization methods in circuit design as [7], [8] in the sense that we also start from a synchronous specification of the system. However, we consider a much higher abstraction level and have a general view on embedded system design like [9]. Our approach is therefore closely related to the work on the polychronous design [10] of embedded systems where [4] the concepts of endo/isochrony have been originally proposed. The definition of isochrony given in [4] is similar to what we call clock-consistency (we use that notion to avoid confusion with other definitions of isochrony like [5], [6] that we also use). However, at least in our setting, the older version of endo/isochrony does not guarantee a correct desynchronization. This can be shown by the following example:

Adders :
$\langle 1 \Rightarrow y_1 = x_1 + x_2 \rangle$
$\langle 1 \Rightarrow y_2 = x_1 + x_3 \rangle$

$\mathsf{behavior}_1$ :

| | | | |
|---|---|---|---|
| $x_1$ | 1 | 2 | ... |
| $x_2$ | 2 | 3 | ... |
| $x_3$ | 4 | 5 | ... |
| $y_1$ | 3 | 5 | ... |
| $y_2$ | 5 | 7 | ... |

$\mathsf{behavior}_2$ :

| | | | |
|---|---|---|---|
| $x_1$ | $\boxdot$ | 1 | 2 |
| $x_2$ | 2 | 3 | ... |
| $x_3$ | 4 | 5 | ... |
| $y_1$ | $\boxdot$ | 4 | ... |
| $y_2$ | $\boxdot$ | 7 | ... |

In this system, two synchronous adders are composed, where one has inputs $x_1$ and $x_2$, and the other one has inputs $x_1$ and $x_3$, thus the *only* shared variable is the input $x_1$. Adders is not clock-consistent, as shown by $\mathsf{behavior}_2$, where in the first reaction, $x_1$ is absent. This makes both adders remain silent, and the inputs on $x_2, x_3$ are lost. However, by the original definition of [4], it is 'isochronous', and it is trivial that both adders are endochronous. Therefore, it should allow a correct desynchronization (by Theorem 2 of [4]). However, it is not the case here, since once the boxes are removed, only $\mathsf{behavior}_1$ can be reconstructed. In later papers [5], [6], the definition of isochrony has been changed to demand the preservation of the flows of the synchronous system. Thus, it moved towards the definition of correct desynchronization, but away from the original intuitive idea of having the *same clock*. Moreover, since based on streams/flows, it is now undecidable. Instead, our result makes use of the original intuition of isochrony, and proves the theorem that was intended in [4], i.e., we prove that given the assumption of endochrony and constructiveness, clock-consistency implies the new definition of isochrony.

### C. Outline

In this paper, we prove a theorem stating that the desynchronization of a synchronous composition of components is correct provided that each component is endochronous, constructive, and that the variables are clock-consistent. We use the notion of clock-consistency as a replacement of isochrony as defined in [5], [6], and therefore come back to its original meaning as intended in [4]. The great advantage over the definition of isochrony given in [5], [6] is that checking clock-consistency is decidable, and can be done by model-checkers using state-based reasoning instead of considering streams or flows of data values.

The remainder of the paper is organized as follows: We first introduce the formal foundations in the next section. We propose our design flow and main theorem in the third section, and discuss some implementation issues in the fourth section, followed by the experimental results. In the final section, we list some conclusions.

## II. Foundations

### A. Synchronous System

The starting point of our design flow is a synchronous system. A *synchronous system* $\langle \mathcal{V}, \mathcal{P} \rangle$ is defined over a set of variables $\mathcal{V} = \mathcal{V}_{in} \cup \mathcal{V}_{loc} \cup \mathcal{V}_{out}$ (input, local and output variables) where each variable $v$ is a pair $(\mathsf{clk}(v), \mathsf{val}(v))$ consisting of its clock $\mathsf{clk}(v)$ and its value $\mathsf{val}(v)$. The domain of clocks is $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ and the domain of other values is $\mathcal{D}$. If the context is clear, we may also simply write $v$ instead of $\mathsf{val}(v)$. $\mathcal{P}$ is a set of synchronous guarded actions (GA for short) over $\mathcal{V}$. Each GA $\rho$ has the form $\langle \gamma \Rightarrow \alpha \rangle$ where the guard $\gamma$ is a boolean formula, and $\alpha$ is an immediate or delayed assignment. For a GA $\rho = \langle \gamma \Rightarrow \alpha \rangle$, we also denote its guard $\gamma$ by $\mathsf{grd}(\rho)$ and its action $\alpha$ by $\mathsf{act}(\rho)$. We denote the set of variables that are read and written by $\rho$ as $\mathsf{rd}(\rho)$ and $\mathsf{wrt}(\rho)$, respectively.

The execution of $\mathcal{P}$ follows the synchronous semantics, i.e., the computation of $\mathcal{P}$ consists of a sequence of discrete reaction steps where in each reaction every $\mathsf{act}(\rho)$ is executed if $\mathsf{grd}(\rho)$ is $\mathsf{true}$. A reaction $r$ is thereby a function: $\mathcal{V} \to (\mathbb{B} \times \mathcal{D})$ that maps variables to values, and we write $[\![e]\!]_r$ for the evaluation of an expression $e$ with respect to $r$. A GA $\rho$ is enabled if $[\![\mathsf{grd}(\rho)]\!]_r = \mathsf{true}$ holds. For an input variable $x$, $r(x)$ depends on the environment, while for local and output variables $x$, we must have $r(x) = (\mathsf{true}, [\![e]\!]_r)$ for every enabled GA $\rho = \langle \gamma \Rightarrow x = e \rangle$, and we must have $r'(x) = (\mathsf{true}, [\![e]\!]_r)$ for every enabled GA $\rho = \langle \gamma \Rightarrow \mathtt{next}(x) = e \rangle$ where $r'$ is the next reaction. Note that an action only assigns values to $\mathsf{val}(x)$ and implicitly sets thereby $\mathsf{clk}(x)$. If no action assigns a value to $x$, $[\![\mathsf{clk}(x)]\!]_r = \mathsf{false}$ and $x$ is *absent* which means it has no value in that reaction.

An execution of a synchronous system is formally defined by a *stream* $t = r_1, r_2, \ldots$ that is an infinite sequence of reactions of $\mathcal{P}$. We denote $r_i$ by $t(i)$, and the set of streams of $\mathcal{P}$ by $\mathcal{T}(\mathcal{P})$. The *projection* $r_{|\mathcal{V}'}$ of reaction $r$ on $\mathcal{V}' \subseteq \mathcal{V}$ is $r_{|\mathcal{V}'} : \mathcal{V}' \to \mathbb{B} \times \mathcal{D}$ where for all $x$ in $\mathcal{V}'$, $r_{|\mathcal{V}'}(x) = r(x)$. $r_{|\{x\}}$ is simply denoted by $r_{|x}$. Projection of a reaction can be extended to streams in the obvious way. A *stuttering* reaction $r$ is a reaction such that for all $v \in \mathcal{V}$, $[\![\mathsf{clk}(v)]\!]_r = \mathsf{false}$. Let $\mathsf{CI}(t)$ be the stream where all stuttering reactions of $t$ are removed. $t_1, t_2$ are *stretch-equivalent* if $\mathsf{CI}(t_1) = \mathsf{CI}(t_2)$, denoted as $t_1 =_{st} t_2$. The leftmost column of Figure 3 shows three synchronous systems that consist of single GAs $\rho_1, \rho_2$ and $\rho_3$. The middle column shows example executions for each of them where $\boxdot$ denotes that the variable is currently absent. Since both $\rho_1$

$$\left[\begin{array}{ll} \rho_1: & \mathcal{V}_{in}=\{x_1\}, \mathcal{V}_{out}=\{x_2\} \\ \mathsf{clk}(x_1) & \Rightarrow \mathtt{next}(x_2)=x_1+1 \\ \rho_2: & \mathcal{V}_{in}=\{x_2\}, \mathcal{V}_{out}=\{x_3\} \\ \mathsf{clk}(x_2) & \Rightarrow \mathtt{next}(x_3)=x_2+1 \\ \rho_3: & \mathcal{V}_{in}=\{x_3\}, \mathcal{V}_{out}=\{x_1\} \\ \mathsf{clk}(x_3) & \Rightarrow x_1=x_3-2 \end{array}\right]$$

| | | |
|---|---|---|
| $t_1$ | $x_1$ | $0 \boxdot 3 \boxdot 6 \boxdot \ldots$ |
| | $x_2$ | $\boxdot 1 \boxdot 4 \boxdot 7 \ldots$ |
| $t_2$ | $x_2$ | $1 \boxdot 4 \boxdot 7 \boxdot \ldots$ |
| | $x_3$ | $\boxdot 2 \boxdot 5 \boxdot 8 \ldots$ |
| $t_3$ | $x_1$ | $0 \boxdot 3 \boxdot 6 \boxdot \ldots$ |
| | $x_3$ | $2 \boxdot 5 \boxdot 8 \boxdot \ldots$ |

| | | |
|---|---|---|
| $t_{1,2}$ | $x_1$ | $0 \boxdot 3 \boxdot 6 \boxdot \ldots$ |
| | $x_2$ | $\boxdot 1 \boxdot 4 \boxdot 7 \ldots$ |
| | $x_3$ | $\boxdot \boxdot 2 \boxdot 5 \boxdot \ldots$ |
| $t_{2,3}$ | $x_1$ | $\boxdot 0 \boxdot 3 \boxdot 6 \ldots$ |
| | $x_2$ | $1 \boxdot 4 \boxdot 7 \boxdot \ldots$ |
| | $x_3$ | $\boxdot 2 \boxdot 5 \boxdot 8 \ldots$ |
| $t_{1,3}$ | $x_1$ | $0 \boxdot 3 \boxdot 6 \boxdot \ldots$ |
| | $x_2$ | $\boxdot 1 \boxdot 4 \boxdot 7 \ldots$ |
| | $x_3$ | $2 \boxdot 5 \boxdot 8 \boxdot \ldots$ |

Fig. 3: Synchronous systems, streams and synchronous composition

and $\rho_2$ have delayed actions, their outputs' clocks are set at the successive reactions of the reactions reading their inputs, which is not the case for $\rho_3$. Another possible execution for $\rho_3$ might be $t_4$ shown below that is stretch-equivalent to $t_3$:

| | | |
|---|---|---|
| $t_4$ | $x_1$ | $0 \boxdot 3 \ 6 \boxdot \ldots$ |
| | $x_3$ | $2 \boxdot 5 \ 8 \boxdot \ldots$ |

The synchronous composition of $\langle \mathcal{V}_1, \mathcal{P}_1 \rangle$ and $\langle \mathcal{V}_2, \mathcal{P}_2 \rangle$, denoted as $\mathcal{P}_1 \| \mathcal{P}_2$, is the synchronous system $\langle \mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{P}_1 \cup \mathcal{P}_2 \rangle$. Two streams $t_1$ and $t_2$ of $\mathcal{P}_1, \mathcal{P}_2$ can be synchronously composed if $t_1|_{\mathcal{V}_1 \cap \mathcal{V}_2} = t_2|_{\mathcal{V}_1 \cap \mathcal{V}_2}$, and the composition is denoted as $t_1 \| t_2$. The right column of Figure 3 shows three streams $t_{1,2}, t_{2,3}$ and $t_{1,3}$ of $\rho_1 \| \rho_2$, $\rho_2 \| \rho_3$ and $\rho_1 \| \rho_3$, respectively.

### B. Clock-Consistency

Pairing $\mathsf{clk}(v)$ and $\mathsf{val}(v)$ for a variable $v$ constrains the semantics of a synchronous system: In particular, if $\mathsf{val}(v)$ is computed or read during some reaction, then its clock $\mathsf{clk}(v)$ must be $\mathsf{true}$. This is called the *clock-consistency* of the synchronous system.

For any reaction $r$ and any variable $v$, we define a predicate $\mathsf{used}(r, v)$ to denote whether $v$ is used by $r$. First, consider a variable $v \in \mathsf{rd}(\mathsf{grd}(\rho))$: If $\mathsf{val}(v)$ is used by an expression, then $\mathsf{used}(r, v) := \mathsf{true}$, otherwise if $\mathsf{clk}(v)$ is used, then $\mathsf{used}(r, v) := [\![\mathsf{clk}(v)]\!]_r$. Second, for $v \in \mathsf{wrt}(\rho)$, if $[\![\mathsf{grd}(\rho)]\!]_r = \mathsf{true}$, then $\mathsf{used}(r, v) = \mathsf{true}$ if $\mathsf{act}(\rho)$ is immediate or if $\mathsf{act}(\rho)$ is delayed then for all successive reaction $r'$, $\mathsf{used}(r', v) = \mathsf{true}$, and for all $v$ in $\mathsf{rd}(\mathsf{act}(\rho))$ $\mathsf{used}(r, v) = \mathsf{true}$. If none of the previous cases applies, it's $\mathsf{false}$.

*Definition 1:* A synchronous system $\langle \mathcal{V}, \mathcal{P} \rangle$ is clock-consistent if and only if for all $x$ in $\mathcal{V}$, for all streams $t$ in $\mathcal{T}(\mathcal{P})$, for all reactions $r$ of $t$, $\mathsf{used}(r, x) \leftrightarrow [\![\mathsf{clk}(x)]\!]_r = \mathsf{true}$.

For example, $\langle \neg \mathsf{clk}(x) \Rightarrow x = 5 \rangle$ is not clock-consistent, as during any reaction, when $\mathsf{clk}(x)$ is $\mathsf{false}$, the guard is satisfied and $x$ is assigned to 5, and thereby $\mathsf{used}(r, x) = \mathsf{true}$. Clock-consistency can be statically checked against the safety condition in the definition. For synchronous data-flow languages like Lustre, it can be embedded and checked in the type system [11].

### C. Constructiveness

One assumption of our design flow is that all synchronous systems must be *constructive* [3] which is checked by most compilers. Informally, a synchronous system is constructive if its outputs can be constructively computed without speculation. Constructiveness ensures that our system is *causally correct* [12], [13] so that our system will be free of causality cycles where components' computations mutually depend on each other at the same instance. *Causality analysis* is performed in order to check if a synchronous system is constructive. Note that constructiveness and clock-consistency are different concepts. For example $\langle \mathsf{clk}(x) \wedge (x = \mathsf{true}) \Rightarrow x = \mathsf{false} \rangle$ is clock-consistent, but is trivially not constructive. More details are discussed in the next section.

## III. The Synchronous Design Flow

Our design flow starts with partitioning a synchronous system $\mathcal{P}$ into a set of synchronous systems $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, so that $\mathcal{P} = \mathcal{P}_1 \| \mathcal{P}_2 \| \ldots \| \mathcal{P}_n$. We call $\mathcal{P}_1, \ldots, \mathcal{P}_n$ the *synchronous components*. In this paper, we only concentrate on the correct desynchronization of the asynchronous composition of $\mathcal{P}_1, \ldots, \mathcal{P}_n$, while the problem of how to determine an efficient partition is not within our scope right now.

Our design flow obeys the criteria for correct desynchronization given in [4], and we briefly review the formal concepts here. To this end we introduce the *desynchronized streams*, or *flows*. Assume $t$ is a stream. Let $k_1, k_2, \ldots$ be the instances where each $[\![\mathsf{clk}(x)]\!]_{t(k_i)} = \mathsf{true}$, and let $s|_x : \mathbb{N} \to \mathcal{D}$ denote the *flow of* $x$ where each $s|_x(i) = [\![\mathsf{val}(x)]\!]_{t(k_i)}$. The flow of $t$ is then $s : \mathcal{V} \to \mathbb{N} \to \mathcal{D}$ where $s(x) = s|_x$ for each $x$ in $\mathcal{V}$, denoted as $\mathsf{Fl}(t)$. Projection $s|_{\mathcal{V}'}$ is thus $\mathcal{V}' \to \mathbb{N} \to \mathcal{D}$. $\mathsf{Fl}$ can be naturally extended to $\mathcal{T}(\mathcal{P})$. Two synchronous streams $t_1$ and $t_2$ are *flow-equivalent* if $\mathsf{Fl}(t_1) = \mathsf{Fl}(t_2)$, denoted by $t_1 =_{fl} t_2$. The asynchronous composition of flows $s_1, s_2$ is denoted by $s = s_1 \|_a s_2$ if $s_1|_{\mathcal{V}_1 \cap \mathcal{V}_2} = s_2|_{\mathcal{V}_1 \cap \mathcal{V}_2}$, where $\forall i \in \{1, 2\}, \forall v \in \mathcal{V}_{s_i}, s(v) = s_i(v)$, for $\mathcal{V}_{s_i}$ the variable domain of $s_i$. $\|_a$ can be naturally extended to sets of flows. We define the asynchronous composition of two synchronous components $\mathcal{P}_1$ and $\mathcal{P}_2$, denoted by $\mathcal{P}_1 \|_a \mathcal{P}_2$, by its behaviors: $\mathsf{Fl}(\mathcal{T}(\mathcal{P}_1)) \|_a \mathsf{Fl}(\mathcal{T}(\mathcal{P}_2))$.

*Definition 2:* A synchronous system $\mathcal{P}$ is endochronous if for streams $t_1$ and $t_2$, $t_1|_{\mathcal{V}_{in}} =_{fl} t_2|_{\mathcal{V}_{in}}$ implies $t_1 =_{st} t_2$.

*Definition 3:* A synchronous system $\mathcal{P}$ is flow-insensitive if for streams $t_1$ and $t_2$, $t_1|_{\mathcal{V}_{in}} =_{fl} t_2|_{\mathcal{V}_{in}}$ implies $t_1 =_{fl} t_2$.

Intuitively, an endochronous system can reconstruct the stretch-equivalent streams from the asynchronous inputs as the synchronous system does, while a flow-insensitive component rebuilds the flow-equivalent streams. Therefore endochronous

systems are flow-insensitive, as stretch-equivalence implies flow-equivalence. The notion of flow-insensitivity generalizes endochrony as well as weak-endochrony [5].

*Definition 4:* [5] The asynchronous composition of synchronous systems $\mathcal{P}_1\|_a \ldots \|_a\mathcal{P}_n$ is isochronous if and only if $\mathsf{Fl}(\mathcal{T}(\mathcal{P}_1))\|_a \ldots \|_a\mathsf{Fl}(\mathcal{T}(\mathcal{P}_n)) = \mathsf{Fl}(\mathcal{T}(\mathcal{P}_1\| \ldots \|\mathcal{P}_n))$.

Isochrony ensures that the asynchronous composition should have exactly the same flows of computations as the synchronous composition, so that the desynchronization won't introduce new flows of behaviors. We examine isochrony in the following subsection in detail.

### A. The Problem

Let's go back to the example in Figure 3. First, all three components are endochronous, since they all have only one input and one output. Therefore flow-equivalence and stretch-equivalence coincide. The Gustave function in Figure 1 is endochronous, since at each reaction step, there is a unique action that can be triggered. Endochrony can be checked statically [4], however endochronous compositions are not necessarily isochronous. Checking isochrony is undecidable [5], and the worse is that isochrony is not compositional, i.e., for systems of $\mathcal{P}_1 \ldots, \mathcal{P}_n$, even if each $\mathcal{P}_i\|_a\mathcal{P}_j$ is isochronous, $\mathcal{P}_1\|_a \ldots \|_a\mathcal{P}_n$ may not be isochronous. These facts are reflected in Figure 3.

First, $\rho_1\|_a\rho_2$, $\rho_2\|_a\rho_3$ and $\rho_1\|_a\rho_3$ are all isochronous. For example, for any flow $s$ of $\rho_1\|_a\rho_2$, we can always separate the two flows $s_1 = s|_{\{x_1,x_2\}}$ and $s_2 = s|_{\{x_2,x_3\}}$. In particular, $s_1$ and $s_2$ share $s|_{x_2}$ which is written by $\rho_1$ and read by $\rho_2$, therefore in the corresponding stream of $\rho_1$, say $t_1$, $\mathsf{clk}(x_2)$ is set by $\rho_1$. We use $t_1|_{x_2}$ as input to $\rho_2$ and assume $t_2$ is produced by $\rho_2$. Since $\rho_2$ is endochronous and $\mathsf{Fl}(t_2|_{x_2}) = s_2|_{x_2}$, it must be $\mathsf{Fl}(t_2) = s_2$, therefore $t_1\|t_2 \in \mathcal{T}(\rho_1\|\rho_2)$ and $\mathsf{Fl}(t_1\|t_2) = s$. Also, for any stream $t$ of $\rho_1\|\rho_2$, it is easy to see that once we split them into $t_1'\|t_2'$, both $\mathsf{Fl}(t_1')$ and $\mathsf{Fl}(t_2')$ can still be computed locally by $\rho_1$ and $\rho_2$ in $\rho_1\|_a\rho_2$. To conclude, by Definition 4 $\rho_1\|_a\rho_2$ is isochronous.

However $\rho_1\|_a\rho_2\|_a\rho_3$ is not isochronous anymore. This can be verified by examining the right column of Figure 3. In particular, according to $\rho_1\|\rho_2$, $x_2$ should be present at the successive reaction of the reaction $x_1$ is read and $x_3$ is present at the successive reaction of the reaction $x_2$ is read. However both $\rho_2\|\rho_3$ and $\rho_1\|\rho_3$ demand $x_1$ and $x_3$ to appear at the same reaction. As a result $\mathcal{T}(\rho_1\|\rho_2\|\rho_3)$ is empty. Nevertheless, $\mathsf{Fl}(t_{1,2})\|_a\mathsf{Fl}(t_{2,3})\|_a\mathsf{Fl}(t_{1,3})$ exists, therefore the asynchronous composition is not isochronous.

### B. The Main Theorem

The counterexample in Figure 3 shows the complexity during desynchronization: neither does the composition of endochronous systems nor mutually isochronous systems lead to an isochronous network. As already discussed in the introduction, in order to avoid creating new behaviors, we must pay respect to the original clocks that coordinate the synchronous components. As a result, for a partitioned system $\mathcal{P} = \mathcal{P}_1\| \ldots \|\mathcal{P}_n$, we demand the following two assumptions:

1 The synchronous system $\mathcal{P}$ is *constructive* and *clock-consistent*.

2 Each system component $\mathcal{P}_i$ is *flow-insensitive*.

Clock-consistency is important not only for ensuring isochronous composition, but also for defining whether a component is flow-insensitive. As an example, $\rho : (\mathsf{true} \Rightarrow o = x + y)$ is clearly constructive, however it is not flow-insensitive considering all inputs. Given the following input: $x : (\mathsf{true}, 3), y : (\mathsf{false}, 6)$, assume $r$ is the reaction to this pair of inputs. As $\mathsf{grd}(\rho)$ is $\mathsf{true}$, $r$ computes $o$, indicating $[\![\mathsf{clk}(y)]\!]_r = \mathsf{true}$. Nevertheless by the input of $y$, $[\![\mathsf{clk}(y)]\!]_r = \mathsf{false}$ and therefore $\mathsf{used}(r, y) \neq [\![\mathsf{clk}(y)]\!]_r$, i.e. $\mathcal{P}$ is not clock-consistent. This example shows the necessity to take inputs into consideration. In particular, in order for $\rho$ to be clock-consistent the environment must push the inputs to it in a friendly way, so that the input variables' clocks are set consistently with the timing they are used.

If we insist that $\rho$ must be clock-consistent, then it is easy to see that the inputs' clock must satisfy $[\![\mathsf{clk}(x)]\!]_r \leftrightarrow [\![\mathsf{clk}(y)]\!]_r$ for all reactions $r$, so that whenever one of the inputs' clock is set, the other's must also be set. When both values appear, the computation is performed. Therefore, it is obviously the case that $\rho$ is flow-insensitive. If we drop the assumption of clock-consistency, $\rho$ would not be flow-insensitive to the following two inputs:

| | | | | |
|---|---|---|---|---|
| $i_1$ | $x$ | 3 | 4 | ... |
| | $y$ | 1 | 5 | ... |

| | | | | |
|---|---|---|---|---|
| $i_2$ | $x$ | $\boxdot$ | 3 | 4 |
| | $y$ | 1 | 5 | ... |

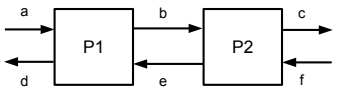
where $i_1$ and $i_2$ are flow-equivalent, as for $i_1$ the first two outputs of $\rho$ are $(\mathsf{true}, 4), (\mathsf{true}, 9)$ but for $i_2$ the first output is $(\mathsf{true}, 8)$. Until now, we can formally present our main theorem:

*Theorem 5:* For a constructive and clock-consistent synchronous system $\mathcal{P} = \mathcal{P}_1\| \ldots \|\mathcal{P}_n$, $\mathcal{P}_1\|_a \ldots \|_a\mathcal{P}_n$ is isochronous if each $\mathcal{P}_i$ is flow-insensitive.

*Proof:* (sketch) By definition of isochrony, we need to prove $\mathsf{Fl}(\mathcal{T}(\mathcal{P}_1))\|_a \ldots \|_a\mathsf{Fl}(\mathcal{T}(\mathcal{P}_n)) \subseteq \mathsf{Fl}(\mathcal{T}(\mathcal{P}_1\| \ldots \|\mathcal{P}_n))$ and $\mathsf{Fl}(\mathcal{T}(\mathcal{P}_1))\|_a \ldots \|_a\mathsf{Fl}(\mathcal{T}(\mathcal{P}_n)) \supseteq \mathsf{Fl}(\mathcal{T}(\mathcal{P}_1\| \ldots \|\mathcal{P}_n))$.

($\supseteq$). Assume $s \in \mathsf{Fl}(\mathcal{T}(\mathcal{P}_1\| \ldots \|\mathcal{P}_n))$, then there must be a stream $\tau \in \mathcal{T}(\mathcal{P}_1\| \ldots \|\mathcal{P}_n)$ such that $s = \mathsf{Fl}(\tau)$. We can extract for each $\mathcal{P}_i$ its own stream $\tau_i = \tau|_{\mathcal{V}_{\mathcal{P}_i}} \in \mathcal{T}(\mathcal{P}_i)$ from $\tau$ and $s_i = \mathsf{Fl}(\tau_i)$. Then it is easily seen that: $s = s_1\|_a \ldots \|_a s_n \in \mathsf{Fl}(\mathcal{T}(\mathcal{P}_1))\|_a \ldots \|_a\mathsf{Fl}(\mathcal{T}(\mathcal{P}_n))$.

($\subseteq$). Without losing generality, we assume $\mathcal{P} = \mathcal{P}_1\|\mathcal{P}_2$ and $s = s_1\|_a s_2 \in \mathsf{Fl}(\mathcal{T}(\mathcal{P}_1))\|_a\mathsf{Fl}(\mathcal{T}(\mathcal{P}_2))$ with the structure:




for the given flows $s_1|_a, s_1|_f$ there are unique flows $s_1'$ and $s_2'$ produced by $\mathcal{P}$. Since $\mathcal{P}$ is causally correct, there are no causal cycles between $\mathcal{P}_1$ and $\mathcal{P}_2$. Therefore, $\mathcal{P}_1$ computes to channel $b$ only depending on inputs from $a$ and previously computed values from $e$. By flow-insensitivity of $\mathcal{P}_1$, a unique flow to $b$ is computed. This is the same case for $\mathcal{P}_2$. Therefore, $s_1' = s_1$ and $s_2' = s_2$. Finally, since $\mathcal{P}$ is clock-consistent, when pushing $s_1|_a$ and $s_2|_f$ to $\mathcal{P}$ in a clock-consistent way, there must be stream $t \in \mathcal{T}(\mathcal{P})$ and $\mathsf{Fl}(t) = s_1\|_a s_2 = s$. ■

A natural application of the main theorem to endochronous components leads to the following corollary:

*Corollary 6:* For a constructive and clock-consistent synchronous system $\mathcal{P} = \mathcal{P}_1 \| \ldots \| \mathcal{P}_n$, if every $\mathcal{P}_i$ is endochronous, then $\mathcal{P}_1 \|_a \ldots \|_a \mathcal{P}_n$ is isochronous.

Corollary 6 follows immediately from Theorem 5 and the fact that endochronous systems are flow-insensitive. Now we can check that the example in Figure 3 cannot be desynchronized, as the synchronous composition $\rho_1 \| \rho_2 \| \rho_3$ is not clock-consistent.

## IV. Implementation Issues

### A. Building Wrappers

By the main theorem, we also need to make sure that each synchronous component is flow-insensitive. Therefore we try to build wrappers for the components ensuring their flow-insensitivity (which might not be always possible). A *wrapper* periodically checks the input channels, and decides based on the observed values whether to trigger a reaction, i.e., to determine the set of *firing rules*. Checking flow-insensitivity is undecidable (otherwise isochrony is decidable), and we check instead for endochrony, which is statically verifiable.

In particular, it is shown in [14] that verifying endochrony of a state-less node in a data-processing network can be reduced to checking if the firing rules describing its behaviors overlap. At each reaction, a firing rule tests the input patterns and based on each pattern triggers a corresponding action. For example, it is easy to see that the firing rules of the Gustave function do not overlap. This technique can be extended to a synchronous component shown as follows.

Given a GA $\rho$, let $\mathsf{term}(\rho)$ be all arithmetic terms occurring in $\rho$ not having the form $\mathsf{clk}(x)$ and let $\mathcal{V}_{\mathsf{t}}^{\rho} = \bigcup_{t \in \mathsf{term}(\rho)} \mathsf{rd}(t)$. Denote $\mathsf{Norm}(\rho)$ be the *normalized formula* of $\rho$ such that for all $t$ in $\mathsf{term}(\rho)$, if it is a conjunct then replace it by $\mathsf{true}$, or if it is a disjunct then replace it by $\mathsf{false}$. The *clock trigger* of $\rho$ is defined as:

$$\mathsf{CT}(\rho) = \mathsf{Norm}(\mathsf{grd}(\rho)) \wedge \bigwedge_{x \in \mathcal{V}_{\mathsf{t}}^{\rho}} \mathsf{clk}(x)$$

We further define:

$$\begin{aligned}
\mathsf{TrigPatt}(\rho) := \{&(\mathcal{G}, \mathcal{V}_{in} \backslash \mathcal{G}) | \mathcal{G} \subseteq \mathcal{V}_{in} \text{ such that there exist:} \\
&\mathcal{I} := \{\mathsf{clk}(x) \mid x \in \mathcal{G}\} \to \{\mathsf{true}\}, \\
&\mathcal{I}' := \{\mathsf{clk}(x) \mid x \in \mathcal{V}_{in} \backslash \mathcal{G}\} \to \{\mathsf{false}\}, \\
&\mathcal{I}'' := \{\mathsf{clk}(x) \mid x \in \mathcal{V} \backslash \mathcal{V}_{in}\} \to \{\mathsf{true}, \mathsf{false}\}, \\
&(\mathcal{I} \cup \mathcal{I}' \cup \mathcal{I}'') \models \mathsf{CT}(\rho)\}, \\
\mathsf{FiringRule}(\rho) := &\langle \mathsf{TrigPatt}(\rho), \mathsf{act}(\rho) \rangle \\
\mathsf{TrigPatt}(\mathcal{P}) := &\{(\mathcal{G}, \mathcal{G}') \mid \exists \rho \in \mathcal{P}.(\mathcal{G}, \mathcal{G}') \in \mathsf{TrigPatt}(\rho)\} \\
\mathsf{FiringRules}(\mathcal{P}) := &\{\mathsf{FiringRule}(\rho) \mid \rho \in \mathcal{P}\}
\end{aligned}$$

We say that the *firing rules* $\mathsf{FiringRules}(\mathcal{P})$ *of system* $\mathcal{P}$ *overlap*, if there exists $(\mathcal{G}, \mathcal{G}')$ and $(\mathcal{F}, \mathcal{F}')$ in $\mathsf{TrigPatt}(\mathcal{P})$ such that either $\mathcal{G} \cap \mathcal{F} \neq \emptyset$ or $\mathcal{G}' \cap \mathcal{F}' \neq \emptyset$. Finally, we utilize the result in [14] and introduce the following theorem:

*Theorem 7:* A synchronous system $\langle \mathcal{V}, \mathcal{P} \rangle$ is endochronous if the firing rules of $\mathcal{P}$ do not overlap.

For example, consider the following synchronous systems:

$$\begin{aligned}
\rho_1 &: (l_1 > 0) \wedge \mathsf{clk}(i_1) \wedge \neg \mathsf{clk}(i_2) \Rightarrow \mathtt{next}(l_1) = f(i_1, i_3) \\
\rho_2 &: (i_1 \leqslant 0) \wedge \neg \mathsf{clk}(i_2) \wedge \mathsf{clk}(i_3) \Rightarrow l_1 = h(i_3) \\
\rho_3 &: \mathsf{clk}(i_1) \vee \mathsf{clk}(i_2) \Rightarrow o = f(l_1, i_3) \\
\rho_4 &: \neg \mathsf{clk}(i_1) \wedge \neg \mathsf{clk}(i_3) \Rightarrow o = f(l_1, i_2)
\end{aligned}$$

where $\mathcal{V}_{in} = \{i_1, i_2, i_3\}, \mathcal{V}_{loc} = \{l_1\}$ and $\mathcal{V}_{out} = \{o\}$, and we have:

$$\begin{aligned}
\mathsf{CT}(\rho_1) &= \big(\mathsf{clk}(i_1) \wedge \neg \mathsf{clk}(i_2)\big) \wedge \big(\mathsf{clk}(l_1) \wedge \mathsf{clk}(i_3)\big) \\
\mathsf{TrigPatt}(\rho_1) &= \{(\{i_1, i_3\}, \{i_2\})\} \\
\mathsf{CT}(\rho_2) &= \big(\neg \mathsf{clk}(i_2) \wedge \mathsf{clk}(i_3)\big) \wedge \big(\mathsf{clk}(i_1)\big) \\
\mathsf{TrigPatt}(\rho_2) &= \{(\{i_1, i_3\}, \{i_2\})\} \\
\mathsf{CT}(\rho_3) &= \big(\mathsf{clk}(i_1) \vee \mathsf{clk}(i_2)\big) \wedge \big(\mathsf{clk}(l_1) \wedge \mathsf{clk}(i_3)\big) \\
\mathsf{TrigPatt}(\rho_3) &= \{(\{i_1, i_3\}, \{i_2\}), (\{i_2, i_3\}, \{i_1\}), (\{i_1, i_2, i_3\}, \emptyset)\} \\
\mathsf{CT}(\rho_4) &= \big(\neg \mathsf{clk}(i_1) \wedge \neg \mathsf{clk}(i_3)\big) \wedge \big(\mathsf{clk}(i_2)\big) \\
\mathsf{TrigPatt}(\rho_4) &= \{(\{i_2\}, \{i_1, i_3\})\}
\end{aligned}$$

For example, for $\rho_3$,

$$\begin{aligned}
\{&\mathsf{clk}(i_1) \mapsto \mathsf{true}, \mathsf{clk}(i_2) \mapsto \mathsf{false}, \mathsf{clk}(i_3) \mapsto \mathsf{true}, \\
&\mathsf{clk}(l_1) \mapsto \mathsf{true}, \ \mathsf{clk}(o) \mapsto \mathsf{true}\}
\end{aligned}$$

is a proper interpretation satisfying $\mathsf{CT}(\rho_3)$ from which $(\{i_1, i_3\}, \{i_2\})$ is extracted in $\mathsf{TrigPatt}(\rho_3)$, and $\rho_1 \| \rho_3$ is not endochronous because $\mathsf{TrigPatt}(\rho_1 \| \rho_3)$ overlaps. In particular, variable pairs in $\mathsf{TrigPatt}(\rho_3)$ already overlap inside. Instead, $\rho_2 \| \rho_4$ is endochronous, since the trigger patterns of $\rho_2$ and $\rho_4$ do not overlap, and $\rho_1 \| \rho_2$ is endochronous, since their trigger patterns coincide.

### B. Explicit Absent Signaling

Once we found $\rho_1 \| \rho_3$ to be not endochronous, we can transmit additional absent signals to make it endochronous. In particular, for $\rho_1 \| \rho_3$, we should explicitly transmit the absent value for $i_1$ and $i_2$ when their clocks are down. This degenerates our implementation to a *latency-insensitive design* [15].

## V. Experimental Results

For illustrating our theorem and related concepts, we propose case studies inspired by [16] where FPGAs are exploited for hardware acceleration of continuous queries against streams of data. A *query plan* is translated from a query language like SQL and is used for computing a data query. In [16], all query plans are implemented purely synchronously, i.e., as synchronous circuits. For better performance, a natural extension is to apply our synchronous design flow to desynchronize the synchronous implementation into an isochronous network.

Our case study is a set of 4 stream query plans for real-time stock-market queries, similar to those in [16]. In particular, $Q_1 = \mathsf{SepBuy} \| \mathsf{SepSell}$ separates the trades into two types: buys and sells, $Q_2 = \mathsf{Avrg}$ and $Q_3 = \mathsf{InBuy} \| \mathsf{InSell} \| \mathsf{Count}$ compute the average prices and total number of trades from two input sequences of trades respectively. Finally $Q_4 = \mathsf{Fork} \| \mathsf{Win}_1 \| \mathsf{Win}_2 \| \mathsf{Merge}$ computes the weighted price of trades regarding a period of time (by using sliding windows). Because of the similarity between the model of computation

TABLE I: Endochronous Components

| | SepBuy | SepSell | Avrg | InBuy | InSell | Count | Fork | $\mathsf{Win}_1$ | $\mathsf{Win}_2$ | Merge |
|---|---|---|---|---|---|---|---|---|---|---|
| Endochrony | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |

TABLE II: Desynchronization Results

| Synchronous Systems | Desynchronization | Assumptions | Signaling | Isochrony |
|---|---|---|---|---|
| $Q_1$ | $\mathsf{SepBuy} \|_a \mathsf{SepSell}$ | ✓ | ✓ | ✓ |
| $Q_3$ | $\mathsf{InBuy} \|_a \mathsf{InSell} \|_a \mathsf{Count}$ | ✓ | ✗ | ✓ |
| $Q_4$ | $\mathsf{Fork} \|_a \mathsf{Win}_1 \|_a \mathsf{Win}_2 \|_a \mathsf{Merge}$ | ✓ | ✓ | ✓ |
| $Q_1 \| Q_2$ | $\mathsf{SepBuy} \|_a \mathsf{SepSell} \|_a \mathsf{Avrg}$ | ✗ | – | ✗ |
| $Q_1 \| Q_3$ | $\mathsf{SepBuy} \|_a \mathsf{SepSell} \|_a \mathsf{InBuy} \|_a \mathsf{InSell} \|_a \mathsf{Count}$ | ✓ | ✗ | ✓ |

of streaming queries (particularly discrete logical time) and synchronous programs, we can easily translate the query plans to synchronous programs and perform our analysis.

Table I shows whether endochrony holds for each query plan's components, where ✓ means the component is endochronous and ✗ means it lacks of endochrony. Based on the results of Table I, we show the result of desynchronization of the query plans in Table II, where the column 'Assumption' lists whether the synchronous composition satisfies the assumptions 1 and 2 of our theorem (✓ for satisfied and ✗ for not satisfied), column 'Signaling' lists whether no additional signaling is needed (✓ for not needed, ✗ for needed, and - for not applicable), and the last column lists whether the decomposition is isochronous.

For example, Count is not endochronous, therefore it needs additional signaling of absent values when desynchronizing $Q_3$. Avrg is assuming that the bought trades and sold trades coming at the same reaction, which is not the case for SepBuy and SepSell (as they separate one single trade stream into two different streams, therefore values of two streams never appear at the same reaction). Therefore $Q_2 \| Q_3$ is not clock-consistent. $Q_4$ uses two copies of a size-2 window to accelerate the weighted price of the two latest trades, therefore Fork outputs two copies of one single trade to both windows at each reaction, and only one of the windows will output a weighted price at each reaction to Merge, and therefore forms a pipeline. All four components of $Q_4$ are endochronous, and the desynchronization is isochronous.

## VI. Conclusions

In this paper, we proved a theorem that states that the desynchronization of a synchronous composition of components is correct provided that each component is endochronous, constructive, and that the variables are clock-consistent. We use the notion of clock-consistency as a replacement of isochrony as defined in [5], [6], and therefore come back to its original meaning as introduced in [4]. The great advantage over the definition of isochrony given in [5], [6] is that checking clock-consistency is decidable, and can be done by model-checkers using state-based reasoning instead of considering streams or flows of data values.

## References


[1] J. Vuillemin, "Correct and optimal implementations of recursion in a simple programming language," *Journal of Computer and System Sciences*, vol. 9, no. 3, pp. 332–354, December 1974.

[2] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[3] G. Berry, "The constructive semantics of pure Esterel," July 1999.

[4] A. Benveniste, B. Caillaud, and P. Le Guernic, "From synchrony to asynchrony," in *Concurrency Theory (CONCUR)*, ser. LNCS, J. Baeten and S. Mauw, Eds., vol. 1664. Eindhoven, The Netherlands: Springer, 1999, pp. 162–177.

[5] D. Potop-Butucaru, B. Caillaud, and A. Benveniste, "Concurrency in synchronous systems," *Formal Methods in System Design (FMSD)*, vol. 28, no. 2, pp. 111–130, 2006.

[6] J.-P. Talpin, J. Ouy, T. Gautier, L. Besnard, and P. Le Guernic, "Compositional design of isochronous systems," *Science of Computer Programming*, vol. 77, no. 2, pp. 113–128, February 2012.

[7] A. Girault and C. Ménier, "Automatic production of globally asynchronous locally synchronous systems," in *Embedded Software (EMSOFT)*, ser. LNCS, A. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491. Grenoble, France: Springer, 2002, pp. 266–281.

[8] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, October 2006.

[9] A. Girault, "A survey of automatic distribution method for synchronous programs," in *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005, pp. 1–20.

[10] S. Suhaib, D. Mathaikutty, S. Shukla, and J.-P. Talpin, "Polychronous methodology for system design: A true concurrency approach," in *International High-Level Design Validation and Test Workshop (HLDVT)*. Monterey, CA, USA: IEEE Computer Society, 2006, pp. 211–214.

[11] J.-L. Colaço and M. Pouzet, "Clocks as first class abstract types," in *Embedded Software (EMSOFT)*, ser. LNCS, vol. 2855. Philadelphia, Pennsylvania, USA: Springer, 2003, pp. 134–155.

[12] T. Shiple, "Formal analysis of synchronous circuits," Ph.D. dissertation, University of California, Berkeley, California, USA, 1996, phD.

[13] K. Schneider, J. Brandt, and T. Schüle, "Causality analysis of synchronous programs with delayed actions," in *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. Washington, District of Columbia, USA: ACM, 2004, pp. 179–189.

[14] Y. Bai, J. Brandt, and K. Schneider, "Monitoring distributed reactive systems," in *High Level Design Validation and Test Workshop (HLDVT)*. Huntington Beach, California, California: IEEE Computer Society, 2012, pp. 84–91.

[15] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, vol. 20, no. 9, pp. 1059–1076, 2001.

[16] R. Müller, J. Teubner, and G. Alonso, "Streams on wires: a query compiler for FPGAs," in *Proceedings of the VLDB Endowment*, vol. 2, no. 1, 2009, pp. 229–240, http://www.vldb.org/pvldb/2/vldb09-622.pdf.