Constructive Polychronous Systems

Jean-Pierre Talpin[1], Jens Brandt[2], Mike Gemünde[2],
Klaus Schneider[2], and Sandeep Shukla[3]

[1] INRIA Rennes-Bretagne-Atlantique, France
[2] Department of Computer Science, University of Kaiserslautern, Germany
[3] Department of Electrical and Computer Engineering, Virginia Tech, USA

Abstract. The synchronous paradigm provides a logical abstraction of time for reactive system design which allows automatic synthesis of embedded programs that behave in a predictable, timely and reactive manner. According to the synchrony hypothesis, a synchronous model reacts to input events and generates outputs that are immediately made available. But even though synchrony greatly simplifies design of complex systems, it often leads to rejecting models when data dependencies within a reaction are ill-specified, leading to causal cycles. Constructivity is a key property to guarantee that the output during each reaction can be algorithmically determined. Polychrony deviates from perfect synchrony by using a partially ordered or relational model of time. It captures the behaviors of (implicitly) multi-clocked data-flow networks and can analyze and synthesize them to GALS systems or to Kahn process networks (KPNs). In this paper, we provide a unified constructive semantic framework, using structural operational semantics, which captures the behavior of both synchronous modules and multi-clocked polychronous processes. Along the way, we define the very first operational semantics of SIGNAL.

1 Introduction

Languages such as ESTEREL [1], QUARTZ [2] or LUSTRE [3] are based on the *synchrony hypothesis*. Synchrony is a logical abstraction of time which greatly facilitates verification and synthesis of safety-critical embedded systems. In particular, it enforces deterministic concurrency, which has many advantages in system design, e.g. avoiding Heisenbugs (i.e. bugs that disappear when one tries to simulate/test them), predictability of real-time behavior, as well as provably correct-by-construction software synthesis [4].

It is also the key to generate *deterministic* single-threaded code from multi-threaded synchronous programs so that synchronous programs can be directly executed on simple micro-controllers without using complex operating systems. Another advantage is the straightforward translation of synchronous programs to hardware circuits [5,6]. Furthermore, the concise formal semantics of synchronous languages allows one to formally reason about program properties [7], compiler correctness and worst-case execution time [8,9].

Under the synchrony hypothesis, computation progresses through totally ordered synchronized execution steps called reactions. The computation involved in reacting to a particular input combination starts by reading the inputs, computing the intermediate values as well as the outputs, and the next state of the system. Each reaction is referred to as a *macro-step*. Computations during the reaction are called *micro-steps*. A reaction is said to happen at a *logical instant*. A logical instant abstracts the duration of a reaction to a single point in a discrete totally ordered timeline.

Consequently, and from a semantic point of view which postulates that a reaction is atomic, neither communication nor computation therefore takes any time in a synchronous instant. Even though this zero-time assumption does not correspond to reality, it is where the power of the synchronous abstraction lies – zero delay is compatible to predictability. If (1) the minimum arrival time of two consecutive values on all inputs is long enough, and if (2) all micro-steps in a reaction (macro-step) are executed according to their data dependencies, then the behaviors under the zero-time assumption are the same as behaviors of the same system in reality.

However, the synchronous abstraction of time also has a drawback. Since outputs are generated in zero-time, and since synchronous systems can typically read their own outputs, there may be cyclic dependencies due to actions modifying their own causes within the same reaction. These issues lead to programs having inconsistent or ambiguous behaviors. In the context of synchronous programs, they are known as *causality problems*, and various solutions have been proposed over the years to tackle them. The most obvious and pragmatic one is to syntactically forbid cyclic data dependencies. It is simple to check but rules out many valid programs. For example, the synchronous language Lustre follows this approach.

Clearly, this is a conservative approach that hardly scales to mapping models on platforms or composing models as this often introduces cycles [10]. Therefore, other synchronous languages, such as Esterel [1], opted for a more sophisticated but costly solution. Their semantics is given in terms of a constructive logic, and compilers perform a causality analysis [11,12,13,14] based on the computation of fixpoints in a three-valued logic similar to Brzozowski and Seger's ternary simulation of asynchronous circuits [15]. Thereby, cyclic dependencies are allowed as long as they can be constructively resolved. This definition of causality does not only show parallels to hardware circuit analysis but also to many other areas.

In contrast to synchronous languages, the polychronous language Signal [16] follows a different model of computation. Execution is not aligned to a totally ordered set of logical instants but to a partial order. This allows one to directly express (abstractions of) asynchronous computations which possibly synchronize intermittently. The lack of a global reference of time offers many advantages for the design of embedded software architectures. First, it is closer to reality since at the system level, integrated components are typically designed based on different clock domains or different paces, which is a desirable feature especially with the advent of, e.g., multi-core embedded processors. Second, polychrony avoids unnecessary synchronization, thereby offering additional optimization

opportunities. Polychrony gives developers the possibility to refine the system in different ways, and compilers can choose from different schedules according to non-functional mapping constraints, which are ubiquitous in embedded systems design. Due to these advantages, SIGNAL is particularly suited as a coordination layer on top of synchronous components to describe a globally asynchronous locally synchronous (GALS) network.

As SIGNAL makes use of the synchronous abstraction of time, it faces the same problems as other synchronous languages. One way to overcome the causality problem is to syntactically forbid cyclic dependencies but as stated before that is not always possible, especially when composing separately specified processes. The SIGNAL compiler uses a so-called *conditional dependence graph* [17,18] to model dependencies between equations and check that all equations in a syntactic cycle cannot happen at the same logical instant. As discussed above, the synchronous languages are all based on slightly varying notions of causality.

This mismatch makes it unnecessarily hard (if not impossible) to integrate, e.g., a set of reactive QUARTZ modules with a SIGNAL data-flow network: should the integration of modules and processes be limited/approximated by syntactically causal data-flow networks, instead of constructive ones ? There is no fundamental reason why a common notion of constructivity should not exist for these languages. So, instead of an approach to causality analysis based on cycle detection, we want to endow SIGNAL with a constructive semantics compatible to that of languages like QUARTZ, which is exactly what this paper presents.

Contribution Our work is rooted in a collaborative project, Polycore, in which we aim at combining the expressive capabilities of the imperative synchronous language QUARTZ and the data-flow polychronous language SIGNAL towards the goal of synthesizing executable GALS systems from the specification of polychronous networks of synchronous modules. This goal demands a common understanding of (i) constructivity and synchronous determinism, best known and studied in the context of imperative synchronous languages, and (ii) endochrony and asynchronous determinism, better developed in the context of relational synchronous languages (yet applicable to imperative ones). This paper tries to bridge the mathematical gap between constructivity and clock/causality analysis in order to show to what extent the former can be explained with the latter.

Our approach consists in the definition of a constructive semantics for polychronous processes that works for synchronous modules as well, so as to share existing notions, theorems and methods. This semantics is of interest on its own: it allows us to better understand the relationship between synchrony and polychrony, between constructivity and endochrony, and to model causality as a formal verification problem. We provide a unified structured operational semantics framework which both captures the synchronous behavior of reactive modules and the multi-clocked behavior of polychronous data-flow networks. This framework allows us to formulate a constructivity theory which captures determinism for both synchronous modules and asynchronous networks. Its expressive capability defines an effective framework in which embedded systems can be designed by combining the best of both styles: imperative modules to describe

system functions and polychronous data-flow networks to describe high-level abstractions of their software architecture. Along the way, we give the very first executable semantics (i.e. an interpreter) of SIGNAL.

Related Work Causal cycles may be real or may be induced by the synchronous abstraction of time. They were first considered in hardware circuits in the early seventies [19,20]. However, causality issues are not only related to the stability analysis of hardware circuits. Berry pointed out that causality analysis is equivalent to theorem proving in intuitionistic (constructive) propositional logic and introduced the notion of constructive programs [11]. Finally, Edwards reformulates the problem such that the existence of dynamic schedules must be guaranteed for the execution of mutually dependent microsteps [13]. Hence, causality analysis is a fundamental algorithm that has already found many applications in computer science. Malik [21] was first to show that this problem in general is NP-hard [21] and used the embedding of Boolean algebra in ternary algebra as proposed by Bryant for the simulation of switch-level circuits [22]. More details about causality analysis for synchronous programs may be found in [21,12,14,23]. In the domain of polychronous programs, causal cycle detection using SMT solvers has been reported in [24,25].

2 Constructive Synchronous Systems

In general, cyclic systems might have no behavior (loss of reactivity), more than one behavior (loss of determinism) or a unique behavior. However, having a unique behavior is generally not sufficient, since there are programs whose unique behavior can only be found by trial and error (or large lookup-tables for all inputs and states, alternatively) – which obviously does not lead to an efficient computation. For this reason, one is usually interested in whether a program has a unique behavior that can be *constructively determined* by the operational semantics. Such a *constructive* semantics is based on fixpoint iteration, which repeatedly execute iterations in order to infer the clocks and values of all signals at every logical instant, i.e., during every reaction.

Embedding Clocks and Values in a Complete Lattice We shall first define some of the essential concepts of fix-point theory used in this paper [26].

Definition 1 (Complete Lattice). *A partial order $(\mathcal{D}, \sqsubseteq)$ is a* lattice*, if every pair $\{x, y\} \subseteq \mathcal{D}$ has a supremum $\bigsqcup\{x, y\}$ and an infimum $\bigsqcap\{x, y\}$ in $\mathcal{D}$. It is* complete *if $\bigsqcup M$ and $\bigsqcap M$ exist for all $M \subseteq \mathcal{D}$. A function $f : \mathcal{D} \to \mathcal{D}$ is monotonic, if for all $x, y \in \mathcal{D}$ s.t. $x \sqsubseteq y$, $f(x) \sqsubseteq f(y)$. It is continuous, if $f(\bigsqcup M) = \bigsqcup f(M)$ and $f(\bigsqcap M) = \bigsqcap f(M)$ holds for all directed sets $M \subseteq \mathcal{D}$.*

To define a constructive semantics, we need to embed the set of data values into a lattice. We achieve this by adding new elements $\mathcal{D}' = \mathcal{D} \cup \{?, \bot, \top, \lightning\}$ to the domain $\mathcal{D}$, the domain of values taken by a variable (see Figure 1).

Starting from $\mathcal{D}'$, we define a partial order $\sqsubseteq \subseteq \mathcal{D}' \times \mathcal{D}'$. Intuitively, the *greater* a value is, the more information we have about it. The error value $\lightning$ is the greatest

element since inconsistent values should never become consistent, while the opposite may occur. We can lift every operator $g : \mathcal{D}^n \to \mathcal{D}^m$ to $g' : \mathcal{D}'^n \to \mathcal{D}'^m$ to evaluate g' in the lattice $\mathcal{D}'$ under the conditions (1) the extended operator g' comply with the original one on $\mathcal{D}$, i. e. $\forall \boldsymbol{x} \in \mathcal{D}^n.\ g'(\boldsymbol{x}) = g(\boldsymbol{x})$ and (2) g' is monotonic w. r. t. $\sqsubseteq$ (for finite $\mathcal{D}'$ this also implies continuity of g').

? signal status is unknown
$\bot$ signal is absent or inhibited
$\top$ signal is present or activated
$\lightning$ signal is inconsistent (e. g. runtime error)

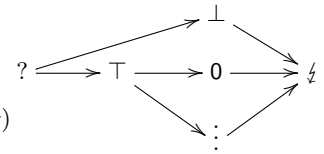

Fig. 1. Embedding a value domain $\mathcal{D}$ in a complete lattice

These conditions predetermine some values of the extension. For the remaining ones, we prefer values $x \in \mathcal{D}$ in order to accept as many programs as possible, and hence choose to use the maximal extension of g w. r. t. $\sqsubseteq$ [23]. As an example, Figure 2, consider Boolean conjunction $\wedge : \mathbb{B}^2 \to \mathbb{B}$. Its extension $\wedge'$ is extended from the original function $\wedge$ on Booleans by choosing the greatest value that satisfies condition 2 above. For example, $\top \wedge' 0$ must be less than or equal $0 \wedge' 0 = 0$ and $1 \wedge' 0 = 0$ (since $\top \sqsubseteq 0$ and $\top \sqsubseteq 1$). As both results are 0, we can also set $\top \wedge' 0 = 0$. Now consider $\top \wedge' 1$, it must be less than or equal to $0 \wedge' 1 = 0$ and $1 \wedge' 1 = 1$. Again, we must set $\top \wedge' 1 = \top$. All the other values in the table can be determined in the same way.

$\wedge'$	?	$\bot$	$\top$	0	1	ϟ
?	?	$\bot$	$\top$	0	$\top$	ϟ
ϟ	ϟ	ϟ	ϟ	ϟ	ϟ	ϟ

$\wedge'$	?	$\bot$	$\top$	0	1	ϟ
$\bot$	$\bot$	$\bot$	ϟ	ϟ	ϟ	ϟ
$\top$	$\top$	ϟ	$\top$	0	$\top$	ϟ

$\wedge'$	?	$\bot$	$\top$	0	1	ϟ
0	0	ϟ	0	0	0	ϟ
1	$\top$	ϟ	$\top$	0	1	ϟ

Fig. 2. Embedding conjunction into $\mathbb{B}'$

We obtain an embedding of all operators into continuous functions over our extended domain $\mathcal{D}'$. As monotonic functions are closed under function composition, the entire system model also yields a monotonic function. Hence, from the Tarski-Knaster theorem 1, it follows that the extensions of all monotonic operators in lattice $\mathcal{D}'$ have fixpoints and, more interestingly, that they have uniquely defined least and a greatest fixpoints. Our framework only needs one half of this theorem, namely the existence and computability of a least fix-point, which requires a complete semi-lattice with an infimum but not necessarily a supremum. Hence, our constructive semantics start with known input variables and local/output variables. If the least fix-point does no longer have unknown values, the program has a unique behavior, it is constructive.

Theorem 1 (Fixpoints in Lattices [26]). *Let $(\mathcal{D}, \sqsubseteq)$ be a complete lattice and $f : \mathcal{D} \to \mathcal{D}$ be a monotonic function. Then, f has fixpoints and the set of fixpoints even has a minimum and a maximum. If f is moreover continuous, then the least fixpoint of f can be computed by the iteration $p_0 := \sqcap\mathcal{D}$, $p_{i+1} := f(p_i)$, and the greatest fixpoint of f by $q_0 := \sqcup\mathcal{D}$, $q_{i+1} := f(q_i)$.*

Constructive Synchronous Guarded Actions In this article, we represent imperative QUARTZ modules using synchronous guarded actions [27,28], as defined in Figure 3 and in the spirit of *guarded commands* [29,30,31], a well-established formalism for the description of concurrent systems. However, our guarded actions follow the *synchronous abstraction of time*. A reactive system is represented by a set of synchronous guarded actions of the form $\langle \gamma \Rightarrow \alpha \rangle$ defined over a set of variables $\mathcal{V}$. The Boolean condition γ is called the guard and α is called the action of the guarded action. An immediate assignment $x = \tau$ writes the evaluated value of τ immediately to the variable x. A delayed assignment $\mathsf{next}(x) = \tau$ stores it until the next execution step.

$$
\begin{array}{llll}
p, q ::= & \mathsf{init}(x) = \tau & \text{(initial)} & \mid \gamma \Rightarrow x = \tau \quad \text{(immediate)} \mid p \parallel q \quad \text{(compose)} \\
\mid & \gamma \Rightarrow \mathsf{next}(x) = \tau & \text{(delayed)} & \mid \mathsf{var}\, x \colon p\, \mathsf{default}\, v \quad \text{(block)} \quad \mid \mathsf{done}\ p \quad \text{(done)}
\end{array}
$$

Fig. 3. Synchronous Guarded Actions

Immediate assignments define a causal dependency within the instant from all the variables read (i. e. variables in the guard γ and on the right-hand side τ) to the written variable x. In contrast, a delayed assignment does not, because they set a values for the following instant. Guarded actions are composed by the operator $p \parallel q$ and grouped by the operator $\mathsf{var}\, x \colon p\, \mathsf{default}\, v$ to locate all actions defining a variable x. If none of these guarded actions apply, x is set to its *default value* v. Immediate variables $\mathcal{E} \subseteq \mathcal{V}$ are reset to their default values (like wires in hardware circuits), while delayed variables $\mathcal{M} \subseteq \mathcal{V}$ are reset to their previous value (like registers in hardware circuits).

As guarded actions manipulate signal (timed) values, we have to keep track of the status and value of each signal. To this end, we use a store $s \in S = X \to \mathcal{D}'$, defined by a function from signal names to status, to evaluate the program expression ϕ with respect to the values stored in s as $s, \phi \rightharpoonup v$. For $\star \in \{\mathsf{and}, \mathsf{or}\}$, we assume that $s, x \star y \rightharpoonup v$ is defined iff $v = s(x) \star s(y) \in \mathbb{B}$ and, for $\star \in \{\mathsf{not}, \mathsf{id}\}$, that $s, \star x \rightharpoonup v$ iff $v = \star s(x) \in \mathbb{B}$. Notice that the relation $s, \phi \rightharpoonup v$ evaluates ϕ to the value $v \in \mathcal{D}$ only if all its free variables are defined on $\mathcal{D}$ in s: it is a synchronous expression (which explains why $\bot$ and $\top$ will not be needed in Figure 4). Additionally, an update operation $\uplus$ sets the status of x in s as follows: $s \uplus (x, v) = s \cup \{(x, \sup(s(x), v))\}$. This definition sets the status of x to $\lightning$ if a status v is assigned that conflicts with that stored in s.

Figure 4 defines transition rules $s, p \rightharpoonup s', q$ for synchronous guarded actions (we assume $v, w, u \in \mathcal{D}$). If the guard γ of an immediate action $\gamma \Rightarrow x = \tau$ evaluates to $\mathsf{1}$ and its action τ to a value $v \in \mathcal{D}$, the store s is updated with $s \uplus (x, v)$. In the case of a delayed action, the additional initial action $\mathsf{init}(x) = v$ is added to be applied in the following instant. The rule for composition $p \parallel q$ defines the possible evaluation schedules of concurrent statements. The marker done indicates that a guarded action has been executed and is propagated by composition and blocks $\mathsf{var}\, x \colon p\, \mathsf{default}\, v$, allowing one to reset the default value of a variable x in that block.

$$\frac{s' = (s(x) \in \mathcal{D})?s, s \uplus (x, v) \quad w = (x \in \mathcal{M})?s'(x), v}{s, \mathsf{var}\, x \colon \mathsf{done}\ (p)\ \mathsf{default}\, v \rightharpoonup s', \mathsf{done}\ (\mathsf{var}\, x \colon p\, \mathsf{default}\, w)} \qquad \frac{s, p \rightharpoonup s', p'}{s,\ q \,|\, p \rightharpoonup s',\ q \,|\, p'}$$

$$\frac{s, \gamma \rightharpoonup 1 \quad s, \tau \rightharpoonup v}{s, \gamma \Rightarrow \mathsf{next}(x) = \tau \rightharpoonup s, \mathsf{done}\ (\mathsf{init}(x) = v \,|\, \gamma \Rightarrow \mathsf{next}(x) = \tau)} \qquad \frac{s, p \rightharpoonup s', p'}{s,\ p \,|\, q \rightharpoonup s',\ p' \,|\, q}$$

$$\frac{s, \gamma \rightharpoonup 1 \quad s, \tau \rightharpoonup v}{s, \gamma \Rightarrow x = \tau \rightharpoonup s \uplus (x, v), \mathsf{done}\ (\gamma \Rightarrow x = \tau)} \qquad \frac{}{s,\ \mathsf{done}\ p \,|\, \mathsf{done}\ q \rightharpoonup s, \mathsf{done}\ (p \,|\, q)}$$

$$\frac{s, \gamma \rightharpoonup 0}{s, \gamma \Rightarrow \mathsf{next}(x) = \tau \rightharpoonup s, \mathsf{done}\ (\gamma \Rightarrow \mathsf{next}(x) = \tau)}$$

$$\frac{s, \gamma \rightharpoonup 0}{s, \gamma \Rightarrow x = \tau \rightharpoonup s, \mathsf{done}\ (\gamma \Rightarrow x = \tau)} \qquad \frac{}{s,\ \mathsf{init}(x) = v \rightharpoonup s \uplus (x, v), ()}$$

Fig. 4. Rules for Synchronous Guarded Actions

3 Polychronous Systems

In contrast to synchronous systems, polychronous specifications [32,16] are based on a partially ordered model of time to express asynchronous computations which possibly need to synchronize sporadically. As the name suggests, polychronous systems use multiple clocks, which means that signals do not need to be present in all instants. Here, the clock of a signal is encoded by its status, present or absent. The value of a signal can only be computed if it is known to be present.

Signal Specifications In the remainder, we use SIGNAL programs to represent polychronous specifications. A SIGNAL program, Figure 5, consists of the composition of several nodes. Each node has an interface consisting of input and output signals and, possibly, local signals. Its body is the composition of equations built from primitive $\star \in \{\mathsf{and}, \mathsf{or}, \mathsf{not}, \$\mathsf{init}, \mathsf{when}, \mathsf{default}\}$ operators.

$$p, q, r ::= \cdots \mid x := y \star z \text{ (equation)}$$

$$\frac{s(x, y, z) \rightharpoonup_\star (a, b, c)}{s,\ x := y \star z \rightharpoonup s \uplus (x, a)(y, b), (z, c),\ x := y \star z}\ (\text{op})$$

$$\frac{s(x, y), a \rightharpoonup_{\$\mathsf{init}} (b, c, d)}{s,\ x := y\, \$\mathsf{init}\, a \rightharpoonup s \uplus (x, b)(y, c),\ x := y\, \$\mathsf{init}\, d}\ (\text{pre})$$

Fig. 5. Small-step operational semantics of polychronous equations

Figure 5 gives the main transition rule for a polychronous equation $x := y \star z$. It relies on the relation $\rightharpoonup_\star$ to check progress from the current status $s(x, y, z)$ of its inputs and output to an hypothetical triple (a, b, c). If so, a transition occurs and the status of (x, y, z) signals is updated. Let us first consider the case of a functional equation such as $x := y\, \mathsf{and}\, z$, Figure 6. From an initial status $s(x, y, z)$, there are three possible ways to progress. First, (1) is when one of the inputs or the outputs is known to be absent, and the others are either unknown or absent (written $?/\bot$). In that case, all three parameters can be deemed absent altogether (right) as they need to occur synchronously. As a result, information on absence may flow backwards from outputs to inputs and possibly inhibits further signals in the environment. A second case is (2) when one of the inputs or the outputs is known to be present, and others either unknown or present

(written ?/⊤). In this case all three parameters can be deemed present (right). As a result information on presence may again flow from outputs to inputs and possibly trigger further input signals in the environment. At last, (3), the values $a, b \in \mathbb{B}$ of inputs are known and the result $a \wedge b$ of the equation is computed.

$\rightharpoonup_{\mathsf{and}}$

$s(x)$	$s(y)$	$s(z)$				
$\bot$	?/⊥	?/⊥	$\bot$	$\bot$	$\bot$	
?/⊥	$\bot$	?/⊥	$\bot$	$\bot$	$\bot$	
?/⊥	?/⊥	$\bot$	$\bot$	$\bot$	$\bot$	(1)
$\top$	?/⊤	?/⊤	$\top$	$\top$	$\top$	
?/⊤	$\top$	?/⊤	$\top$	$\top$	$\top$	(2)
?/⊤	?/⊤	$\top$	$\top$	$\top$	$\top$	
?/⊤	a	?/⊤	$\top$	a	$\top$	(3)
?/⊤	?/⊤	a	$\top$	$\top$	a	
?/⊤	b	a	$a \wedge b$	b	a	

$\rightharpoonup_{\mathsf{when}}$

$s(x)$	$s(y)$	$s(z)$				
$\bot$	?/⊥	?/⊥	$\bot$	$\bot$	$\bot$	
?/⊥	$\bot$	?/⊥	$\bot$	$\bot$	$\bot$	(1)
?/⊥	?/⊥	$\bot$	$\bot$	$\bot$	$\bot$	
?/⊥	x	0	$\bot$	x	0	
$\top$	?/⊤	a	$\top$	$\top$	$a \sqcup 1$	(2)
?/⊤	a	1	a	a	1	

Fig. 6. Small-step relations of polychronous equations

The same evaluation principle can be applied to the downsampling operator $x := y \,\mathsf{when}\, z$, Figure 6, the rules are the same for propagating absence (1) and there is only one possible way to propagate presence: when that of the output is already known, for any positive progress $a, b \in \{?, \top, 0, 1\}$ of the inputs. When $z = 0$, we "don't care" the first input: $x \in \mathcal{D}'$, and issue the output as absent. Again, there is only one way to propagate presence (2): when that of the output is already known and for any positive progress $a \in \{?, \top, 1\}$ of the inputs. Also, there is only one case where the output can possibly have a value $a \in \mathbb{B}$, when $z = 1$ (supremum $a \sqcup 1$).

$\rightharpoonup_{\mathsf{default}}$

$s(x)$	$s(y)$	$s(z)$				
$\bot$	?/⊥	?/⊥	$\bot$	$\bot$	$\bot$	
?/⊥	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	(1)
?	$\top$	x	$\top$	$\top$	x	
?	x	$\top$	$\top$	x	$\top$	(2)
?/⊤	a	x	a	a	x	
?/⊤	$\bot$	a	a	$\bot$	a	(3)

$\rightharpoonup_{\mathsf{\$init}}$

$s(x)$	$s(y)$	a				
$\bot$	?/⊥	a	$\bot$	$\bot$	a	
?/⊥	$\bot$	a	$\bot$	$\bot$	a	(1)
$\top$	?/⊤	a	a	$\top$	a	
?/⊤	$\top$	a	a	$\top$	a	(2)
?/⊤	b	a	a	b	b	
a	b	a	a	b	b	(3)

Fig. 7. Small-step relations of polychronous equations

The merge operator $x := y \,\mathsf{default}\, z$ works in a way opposite to sampling, Figure 6, there is only one way its result (or its inputs) can be ruled absent (1). There are many ways merge can make positive progress from the knowledge of either of its inputs, "regardless" of the (don't care) value $x \in \mathcal{D}'$ of the other (2). Finally, merge gives a value $a \in \mathcal{D}$ to its output if y holds a or when it is absent and z does (3). The case of the delay equation $x := y \,\mathsf{\$init}\, a$ is a little trickier and first requires a specific rule (pre) to take care of the fact that its third argument, the state, is a value $a, d \in \mathcal{D}$. Apart from that, it mainly acts

as a synchronous operation between its input and output signals. Once again, absence can be propagated both ways in the data-flow to either inhibit or trigger signals in the environment (1). However, as soon as one of the input or output is known to be present, delay forwards its stored value $a \in \mathcal{D}$ to the output (2). Last, once the input value $b \in \mathcal{D}$ has been calculated (3), it can be stored in place of the old one (here a, the output).

Example We consider the specification of a polychronous counter of input n and output o. Every time its execution is triggered, its purpose is to provide the value of a count along with its output signal o. When that count reaches 0, the counter synchronizes with the input signal n to reset the count with a new value. Clock synchronization $n \,\mathsf{sync}\, y$ is rewritten by an equation $(z := ((y = y) = (n = n)))/z$ that forces y and n to have the same status.

$$counter(n, o) \triangleq (c := o \,\$\mathsf{init}\, 0 \,|\, o := n \,\mathsf{default}\, x \,|\, x := (c - 1) \,|\, n \,\mathsf{sync}\, y \,|\, y := 1 \,\mathsf{when}\, (c = 0)))\,/cxy$$

In the remainder, and referring to a QUARTZ module or a SIGNAL process $\langle s, p \rangle$, we note $V(p)$ for all signals of p, $O(p)$ for its output signals, $I(p)$ for its inputs and $L(p)$ for its locals. The counter has input n, output o and locals $\{c, x, y\}$. The local signal c is the current count, x its decrement, and y the reset condition. The execution of the counter is modeled by a series of steps (changes are marked with $_\bullet$) in which we assume c is initially 1 by $c := o \,\$\mathsf{init}\, 1$ (to model the operation of decrementing the counter). First, the environment triggers execution by setting the output signal o to present. This forces the current count c evaluate ($\rightharpoonup_{\$\mathsf{init}}$). Then, its decrement x and the reset condition y can both be evaluated ($\rightharpoonup_{sub}$ and $\rightharpoonup_{eq}$). Next, the status of the input n can be defined from that of y ($\rightharpoonup_{\mathsf{sync}}$). Since it is absent and c is present with the count, the output o can now be output ($\rightharpoonup_{\mathsf{default}}$) and its value is stored in place of the previous one ($\rightharpoonup_{\$\mathsf{init}}$). As the example shows, we obtain a constructive and executable operational semantics that can capture the behavior of synchronous QUARTZ modules as well as of polychronous SIGNAL processes.

$$\begin{array}{rll}
(c,?)(n,?)(o,\top_\bullet)(x,?)(y,?) \rightharpoonup (c,1_\bullet)\ (n,?)\ (o,\top)\ (x,?)\ (y,?) & \text{from } \rightharpoonup_{\$\mathsf{init}} \\
\rightharpoonup (c,1)\ (n,?)\ (o,\top)(x,0_\bullet)\ (y,?) & \text{from } \rightharpoonup_{\mathsf{sub}} \\
\rightharpoonup (c,1)\ (n,?)\ (o,\top)\ (x,0)\ (y,\bot_\bullet) & \text{from } \rightharpoonup_{\mathsf{when}} \\
\rightharpoonup (c,1)\ (n,\bot_\bullet)(o,\top)\ (x,0)\ (y,\bot) & \text{from } \rightharpoonup_{\mathsf{sync}} \\
\rightharpoonup (c,1)\ (n,\bot)\ (o,0_\bullet)\ (x,0)\ (y,\bot) & \text{from } \rightharpoonup_{\mathsf{default}} \\
\rightharpoonup (c,1)\ (n,\bot)\ (o,0)\ (x,0)\ (y,\bot) & \text{from } \rightharpoonup_{\$\mathsf{init}}
\end{array}$$

From Synchrony to Asynchrony Our next step is to embed this semantics in its execution environment of asynchronous streams in order to reason about networked processes. Towards this goal, we first need to interface the small step operational semantics with an environment of asynchronous streams. We represent a stream by a word $w \in \mathbb{S} = \mathcal{D}^*$ of values a and define the operation of reading a from a stream as $a.w$ and writing onto it as $w.a$ in order to reflect a first-in-first-out protocol. The environment or trace of a process p in a network is represented by a finite map $E \in \mathbb{T} = X \mapsto \mathbb{S}$ that associates signal names x

and streams w. Since a module p owns a local store s, we shall note $\langle s, p\rangle$ its embedding in a network P constructed by asynchronous composition.

$$w ::= \epsilon \mid a.w \mid w.a \quad \text{(FIFO)} \qquad P, Q ::= \langle s, p\rangle \mid P \parallel Q \quad \text{(network)}$$

Then, we simply lift the transition relation $s, p \rightharpoonup t, q$ to further account for the way a process p interacts with asynchronous environment and construct the small-step semantics $E, P \rightharpoonup F, Q$ to define this interaction between local, synchronous, steps of execution and shared, asynchronous, FIFO streams, i.e.,

$$\frac{s, p \rightharpoonup t, q}{E, \langle s, p\rangle \rightharpoonup E, \langle t, q\rangle}$$

Interfacing polychronous processes The execution of a SIGNAL process p is locally triggered by activating the status of one or several of its signals, by setting them present (Figure 8, top-left). We call these signals $T(p)$ – the triggers of p. Once a trigger is enabled, other signals can be and, when an input x is, its value can be loaded from the environment E (right). The output of a process onto streams is performed when computation within the process is completed (Figure 8, bottom) in order to respect the synchronous step paradigm. The "flush" relation $E, \langle s, p\rangle \rightharpoondown F, \langle s, p\rangle$ models this very step. It is defined by pulling computed output values from the local store to the shared streams. It delimits the temporal barriers of a synchronous instant or reaction and filters inputs which have been read and signals which are absent.

$$\frac{x \in T(p)}{E, \langle s \uplus (x, ?), p\rangle \rightharpoonup E, \langle s \uplus (x, \top), p\rangle} \qquad \frac{x \in I(p)}{E \uplus (x, a.w), \langle s \uplus (x, \top), p\rangle \rightharpoonup E \uplus (x, w), \langle s \uplus (x, a), p\rangle}$$

$$\frac{x \notin O(p) \vee a \notin \mathcal{D}}{E, \langle s \uplus (x, a), p\rangle \rightharpoondown E, \langle s \uplus (x, ?), p\rangle} \qquad \frac{x \in O(p) \wedge a \in \mathcal{D}}{E \uplus (x, w), \langle s \uplus (x, a), p\rangle \rightharpoondown E \uplus (x, w.a), \langle s \uplus (x, ?), p\rangle}$$

Fig. 8. Interface semantics of polychronous equations p

Interfacing synchronous modules The asynchronous interface of a synchronous QUARTZ module is simpler. It only requires rules for the $\rightharpoondown$ relation to handle writing the computed output values and reading the next values of inputs (Figure 9, s_x stands for s without x).

$$\begin{aligned} E \uplus (x, w), \langle s \uplus (x, a), p\rangle &\rightharpoondown E \uplus (x, w.a), \langle s_x \uplus (x, ?), p\rangle && x \in O(p) \\ E \uplus (x, a.w), \langle s, p\rangle &\rightharpoondown E \uplus (x, w), \langle s_x \uplus (x, a), p\rangle && x \in I(p) \\ E, \langle s, p\rangle &\rightharpoondown E, \langle s_x \uplus (x, ?), p\rangle && x \in L(p) \\ E, \langle s, \mathsf{done}\ p\rangle &\rightharpoondown E, \langle s, p\rangle \end{aligned}$$

Fig. 9. Interface semantics of synchronous modules p

A constructive GALS semantics Finally, the GALS semantics $E, P \to E, Q$, Figure 10, can be defined for synchronous modules and polychronous processes. It

comprises of local synchronous execution steps $E, P \rightharpoonup E, Q$ (left) during which inputs are read, outputs are computed, and a synchronization step $E, P \rightharpoonup^* F, Q$ (middle), during which outputs are registered in the environment. and the local store is reset to start a new step of execution. Interleaving or scheduling is again defined by parallel composition (right).

$$\frac{E, P \rightharpoonup E, Q}{E, P \to E, Q} \quad \frac{E, \langle s, p\rangle \rightharpoonup^* F, \langle s', q\rangle}{E, \langle s, p\rangle \to F, \langle s', q\rangle} \quad \frac{E, P \to F, R}{E, P \parallel Q \to F, R \parallel Q} \quad \frac{E, P \to F, R}{E, P \parallel Q \to F, P \parallel R}$$

Fig. 10. Interface semantics of synchronous modules p

Communication from a process to another is assumed to be point-to-point. Multiple writers on a single stream are not allowed as per the synchronous paradigm. Broadcast communication can be simulated by multiplexing the output of the writer to multiple readers: for any $x \in dom(E)$ such that $x \in O(P)$ and $x \in I(Q)$ and $x \in I(R)$, we can rewrite $P \parallel Q \parallel R$ as $P \parallel (Q[y/x] \parallel R[z/x] \parallel \langle (), y := x \,|\, z := x\rangle)\,/yz$ using any y, z not in Q, R.

4 Determinism and constructivity

The layered constructive semantics allows us to formulate classical properties of polychronous processes, as stated in the trace or logical settings of [33], yet in a constructive operational semantics framework. We first state the property of reactivity pertaining to the correctness of Quartz programs. A synchronous module p is reactive iff for any combination of input values, its transition function always terminates by producing a combination of output values.

Definition 2 (Reactivity). *A module p is reactive iff, for all valuation $s_0 = \{(x, a) \,|\, x \in I(p), a \in \mathcal{D}\} \cup \{(y, ?) \,|\, y \in V(p) \setminus I(p)\}$, there exists $s_0, p \rightharpoonup^* s, q$ with s defined on $\mathcal{D}$.*

Synchronous determinism relates to reactivity. However, while reactivity assumes that the values of all inputs are known, synchronous determinism also applies to the case of a Signal process, whose inputs may not all be read: a process p is synchronously deterministic iff for any initial status s, its step relation always converges to a unique fixpoint.

Definition 3 (Synchronous Determinism). *A process p is synchronously deterministic iff for all store s and derivations $s, p \rightharpoonup^* t, q$ and $s, p \rightharpoonup^* u, r$ we have $t = u$ and $q = r$.*

Now, once embedded in an asynchronous environment of streams E, determinism becomes endochrony or asynchronous determinism. A process P is asynchronously deterministic iff for every input trace E, it yields a unique output trace F and unique final state Q.

Definition 4 (Asynchronous Determinism). *A process p is asynchronously deterministic iff for all input traces E and derivations $E, \langle s, p\rangle \rightharpoonup^* F, \langle t, q\rangle \rightarrow^* G, \langle u, r\rangle$ and $E, \langle s, p\rangle \rightharpoonup^* F', \langle t', q'\rangle \rightarrow^* G', \langle u', r'\rangle$, $G = G'$ and $r = r'$.*

Notice that, in the case of a deterministic process, equality $G = G'$ is defined over finite maps between variables and values and that, in $r = r'$, syntactic congruence is taken care of by the rules of synchronous composition i.e. $G = G'$ and $r = r'$ denote canonical configurations).

Constructivity So far, and thanks to the definition of a complete domain of clocked signals, we have defined the very first, executable, structured operational semantics for the polychronous data-flow language SIGNAL. Its purpose starts to unveil as we consider the fixpoint theoretic implication of its definition on that continuous domain and try to formulate the property of constructivity. Originally, constructivity was defined as a property of an imperative synchronous module that pertains to the reachability of a stable state in its electrical semantics [12]. In the operational setting of the QUARTZ language, this means that, given any combination of input values, a synchronous module p should always define unique output values.

Definition 5 (Synchronous constructivity). *A module p is synchronous constructive iff for all initial valuations $s_0 = \{(x, a) \mid x \in I(p), a \in \mathcal{D}\} \cup \{(y, ?) \mid y \in V(p) \setminus I(p)\}$, $s_0, p \rightharpoonup^* s, q$ and s is defined on $\mathcal{D}$ (i.e. $s = lfp{\rightharpoonup_p}(s_0)$).*

Thanks to the fidelity level of our small-step operational framework, the formulation of constructivity matches that of reactivity and determinism as stated in the previous section.

Proposition 1. *If p is synchronously constructive then p is reactive and synchronously deterministic*

We shall now formulate constructivity in the context of a polychronous process p. However, it is clear from the example of the counter that a polychronous process not necessarily is constructive in the sense as QUARTZ: it may not be reactive.

$$counter(n, o) \triangleq (c := o\,\$\mathsf{init}\,0 \mid o := n\,\mathsf{default}\,x \mid x := (c-1) \mid n\,\mathsf{sync}\,y \mid y := 1\,\mathsf{when}\,(c = 0)))\,/cxy$$

Unlike a QUARTZ module, the counter triggers a sequence of execution steps every time its trigger o is activated. It only loads an input from n when c is 0. Hence, it is not reactive w. r. t. its input signals, but it is reactive w. r. t. its input streams. This observation yields a more general (asynchronous) formulation of constructivity: a process p is asynchronously constructive iff for any combination of values available from its input streams, it always produces an output.

Definition 6 (Asynchronous constructivity). *A process p is asynchronously constructive iff for any environment E of non-empty streams defined on $V(p)$ and $s_0 = \{(x, ?) \mid x \in V(p)\}$, we have $E, \langle s_0, p\rangle \rightharpoonup^* F, \langle s, q\rangle$ with s defined on $\mathcal{D}^\perp$ (i.e. $(F, s) = lfp{\rightharpoonup_p^*}(E, s_0)$).*

Notice that the transitive closure $\rightharpoonup^* F$ of a constructive process always yields a unique valuation on $\mathcal{D}^\perp$ (it is a continuous domain). If a process isn't constructive, it either blocks (e.g. $x = y \,|\, y = x$) and yields values below $\mathcal{D}^\perp$ or conflicts (e.g. $x = 0 \,|\, x = 1$) and yields value $\lightning$. Again, asynchronous constructivity corresponds to the property of asynchronous determinism in the logical and denotational frameworks of SIGNAL [33,34].

Proposition 2. *If p is asynchronously constructive then p is asynchronously deterministic*

Case of isochronous systems Definition 6 of asynchronous constructivity is formulated in a way that may seem to coincide with a class of systems which can deterministically be executed starting from a singleton trigger $T(p)$. This case indeed characterizes so called endochronous systems [33] whose input/output signal status can all be decided from that of a single one, "the master clock". A larger class of deterministic systems can be captured by considering processes p that deterministically execute from several, independent, concurrent triggers: so-called weakly endochronous systems [34]. In our framework, a weakly endochronous process p is characterized by a set of multiple, independent triggers $T(p)$, each of them triggers execution of independent computations and yields a confluent state. To allow this, however, we additionally need to allow some of these triggers to possibly (non-deterministically) be chosen to be absent during a given execution step (all computations depending of that trigger would then evaluate to absent). This can be done by choosing the following inhibition rule, instead of the triggering one:

$$\frac{x \in T(p)}{E, \langle s \uplus (x, ?), p\rangle \rightharpoonup E, \langle s \uplus (x, \perp), p\rangle}$$

While a weakly endochronous process may have several triggers, it is additionally required to be stuttering invariant: an inhibited process shall not react to absence.

Definition 7 (Stuttering). *p is stuttering iff for $s = \{(x, \perp) \,|\, x \in T(p)\} \cup \{(x, ?) \,|\, x \in V(p) \setminus T(p)\}$, we have $\langle s, p\rangle \rightharpoonup^* \langle s', p\rangle$ and $s' = \{(x, \perp) \,|\, x \in V(p)\}$*

Notice that Definition 6 accommodates the case of weakly endochronous systems with the above additions of a rule and of a definition for stuttering.

5 Summary

In this article, we defined a constructive operational semantics to unite the synchronous imperative language QUARTZ and the polychronous data-flow language SIGNAL in a common framework. This model is of interest on its own, since it allows us to better understand the relationship between synchrony and polychrony, between constructivity and endochrony. It additionally allows us to model the causality problem as a formal verification problem. We formulated a constructivity theory which captures the behavior of correct synchronous modules and deterministic asynchronous/polychronous networks. Along the way, we provided the very first truly executable operational semantics of SIGNAL.

Acknowledgement This work is partially supported by INRIA under associate-project Polycore, by the US Air Force Research Laboratory, grant FA8750-11-1-0042, and the German Research Foundation (DFG).

References

1. Berry, G.: The foundations of Esterel. In Plotkin, G., Stirling, C., Tofte, M., eds.: Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press (1998) 425–454
2. Schneider, K.: The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern (December 2009)
3. Halbwachs, N.: A synchronous language at work: the story of Lustre. In: Formal Methods and Models for Codesign (MEMOCODE), IEEE Computer Society (2005) 3–11
4. Titzer, B., Palsberg, J.: Nonintrusive precision instrumentation of microcontroller software. In Paek, Y., Gupta, R., eds.: Languages, Compilers, and Tools for Embedded Systems (LCTES), Chicago, Illinois, USA, ACM (2005) 59–68
5. Berry, G.: A hardware implementation of pure Esterel. Sadhana **17**(1) (March 1992) 95–130
6. Rocheteau, F., Halbwachs, N.: Pollux: A Lustre-based hardware design environment. In Quinton, P., Robert, Y., eds.: Algorithms and Parallel VLSI Architectures II, Gers, France, Elsevier (1992) 335–346
7. Schneider, K., Brandt, J., Schuele, T.: A verified compiler for synchronous programs with local declarations. Electronic Notes in Theoretical Computer Science **153**(4) (2006) 71–97
8. Logothetis, G., Schneider, K.: Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In: Design, Automation and Test in Europe (DATE), IEEE Computer Society (2003) 10196–10203
9. Boldt, M., Traulsen, C., von Hanxleden, R.: Compilation and worst-case reaction time analysis for multithreaded Esterel processing. EURASIP Journal on Embedded Systems (2008)
10. Stok, L.: False loops through resource sharing. In: International Conference on Computer-Aided Design (ICCAD), IEEE Computer Society (1992) 345–348
11. Berry, G.: The constructive semantics of pure Esterel. `http://www-sop.inria.fr/meije/esterel/esterel-eng.html` (July 1996)
12. Shiple, T., Berry, G., Touati, H.: Constructive analysis of cyclic circuits. In: European Design Automation Conference (EDAC), Paris, France, IEEE Computer Society (1996) 328–333
13. Edwards, S.: Making cyclic circuits acyclic. In: Design Automation Conference (DAC), Anaheim, California, USA, ACM (2003) 159–162
14. Schneider, K., Brandt, J., Schuele, T.: Causality analysis of synchronous programs with delayed actions. In: Compilers, Architecture, and Synthesis for Embedded Systems (CASES), Washington, District of Columbia, USA, ACM (2004) 179–189
15. Brzozowski, J., Seger, C.J.: Asynchronous Circuits. Springer (1995)
16. Le Guernic, P., Gauthier, T., Le Borgne, M., Le Maire, C.: Programming real-time applications with SIGNAL. Proceedings of the IEEE **79**(9) (1991) 1321–1336
17. Le Guernic, P., Benveniste, A.: Real-time, synchronous, data-flow programming: The language SIGNAL and its mathematical semantics. Research Report 533, INRIA (June 1986)

18. Besnard, L., Gautier, T., Le Guernic, P., Talpin, J.P.: Compilation of polychronous data flow equations. In Shukla, S., Talpin, J.P., eds.: Synthesis of Embedded Software – Frameworks and Methodologies for Correctness by Construction. Springer (2010)
19. Kautz, W.: The necessity of closed circuit loops in minimal combinational circuits. IEEE Transactions on Computers (T-C) **C-19**(2) (February 1970) 162–166
20. Rivest, R.: The necessity of feedback in minimal monotone combinational circuits. IEEE Transactions on Computers (T-C) **C-26**(6) (1977) 606–607
21. Malik, S.: Analysis of cycle combinational circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD) **13**(7) (July 1994) 950–956
22. Bryant, R.: A switch level model and simulator for MOS digital systems. IEEE Transactions on Computers (T-C) **C-33**(2) (February 1984) 160–177
23. Schneider, K., Brandt, J., Schuele, T., Tuerk, T.: Maximal causality analysis. In Desel, J., Watanabe, Y., eds.: Application of Concurrency to System Design (ACSD), Saint-Malo, France, IEEE Computer Society (2005) 106–115
24. Jose, B., Gamatie, A., Ouy, J., Shukla, S.: SMT based false causal loop detection during code synthesis from polychronous specifications. In Singh, S., ed.: Formal Methods and Models for Codesign (MEMOCODE), Cambridge, UK, IEEE Computer Society (2011) 109–118
25. Nanjundappa, M., Kracht, M., Ouy, J., Shukla, S.: Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework. In: Electronic System Level Synthesis Conference (ESLsyn), 2012. (june 2012) 1 –6
26. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**(2) (1955) 285–309
27. Brandt, J., Schneider, K.: Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (March 2011)
28. SYNCHRON: The common format of synchronous languages - the declarative code DC. Technical report, C2A-SYNCHRON project (1998)
29. Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM (CACM) **18**(8) (1975) 453–457
30. Chandy, K., Misra, J.: Parallel Program Design. Addison-Wesley (May 1989)
31. Dill, D.: The Murphi verification system. In Alur, R., Henzinger, T., eds.: Computer Aided Verification (CAV). Volume 1102 of LNCS., New Brunswick, New Jersey, USA, Springer (1996) 390–393
32. Gamatié, A., Gautier, T., Le Guernic, P., Talpin, J.: Polychronous design of embedded real-time applications. ACM Transactions on Software Engineering and Methodology (TOSEM) **16**(2) (April 2007)
33. Le Guernic, P., Talpin, J.P., Le Lann, J.C.: Polychrony for system design. Journal of Circuits, Systems, and Computers (JCSC) **12**(3) (June 2003) 261–304
34. Potop-Butucaru, D., Sorel, Y., de Simone, R., Talpin, J.P.: Correct-by-construction asynchronous implementation of modular synchronous specifications. Fundamenta Informaticae **108**(1-2) (2011) 91–118