Constructive Polychronous Systems☆

Jean-Pierre Talpin[a], Jens Brandt[b], Mike Gemünde[b], Klaus Schneider[b],
Sandeep Shukla[c]

[a] *INRIA Rennes-Bretagne-Atlantique, France*
[b] *University of Kaiserslautern, Germany*
[c] *Virginia Tech, USA*

Abstract

The synchronous paradigm provides a logical abstraction of time for reactive system design which allows automatic synthesis of embedded systems that behave in a predictable, timely, and reactive manner. According to the synchrony hypothesis, a synchronous model reacts to inputs by generating outputs that are immediately made available to the environment. While synchrony greatly simplifies the design of complex systems in general, it can sometimes lead to causal cycles. In these cases, constructiveness is a key property to guarantee that the output of each reaction can still be always algorithmically determined.

Polychrony deviates from perfect synchrony by using a partially ordered, i.e., a relational model of time. It encompasses the behaviors of (implicitly) multi-clocked data-flow networks of synchronous modules and can analyze and synthesize them as GALS systems or Kahn process networks (KPNs).

In this paper, we present a unified constructive semantic framework using structured operational semantics, which encompasses both the constructive behavior of synchronous modules and the multi-clocked behavior of polychronous networks. Along the way, we define the very first executable operational semantics of the polychronous language Signal.

1. Introduction

Languages such as Esterel [1], Quartz [2] or Lustre [3] are based on the *synchrony hypothesis* [4, 5]. 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

☆This work is partially supported by INRIA associate-project Polycore, by the Deutsche Forschungsgemeinschaft DFG, by the US Air Force Research Laboratory (grant FA8750-11-1-0042) and the US Air Force Office for Scientific Research (grant FA8655-13-1-3049).

Email addresses: `Jean-Pierre.Talpin@inria.fr` (Jean-Pierre Talpin), `brandt@cs.uni-kl.de` (Jens Brandt), `gemuende@cs.uni-kl.de` (Mike Gemünde), `schneider@cs.uni-kl.de` (Klaus Schneider), `shukla@vt.edu` (Sandeep Shukla)

tries to simulate/test them), predictability of real-time behavior, as well as provably correct-by-construction software synthesis [6].

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 [7] that allows one to use synchronous languages in HW/SW co-design. Furthermore, the concise formal semantics of synchronous languages allows one to formally reason about program properties [8], compiler correctness and worst-case execution time [9, 10].

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 and the outputs values of the reaction, as well as the next state of the system. Each complete reaction is referred to as a *macro-step* whereas computations during the reaction are called *micro-steps*. A reaction is said to happen at a *logical instant* that 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 physical 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 inter-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 the 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 may 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 which is simple to check but rules out many valid programs. For example, the synchronous data-flow language Lustre follows this approach [11]. More powerful algorithms for causality analysis are discussed in Section 2. One key advantage of Lustre is that it corresponds to a strict subclass of Kahn networks in which all actors synchronise on every execution step, so called synchronous Kahn networks [12]. Kahn networks are none to have a compositional asynchronous semantics [13].

In contrast to synchronous languages, the polychronous language Signal [14] follows a different model of computation. Execution is not aligned to a totally ordered set of logical instants but to a partially ordered model of time. 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.

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* [15, 16, 17] to model dependencies between equations and to 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 currently 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 constructiveness should not exist for these languages. So, instead of an approach to causality analysis based on syntactic 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.

2. Related Work

As mentioned in the introduction, causality analysis performed in data-flow languages is a conservative approach to checking the constructiveness of a system. However, mapping or composing models on platforms often requires the introduction of pseudo-cycles [18, 19, 20]. Therefore, other synchronous languages like Esterel [1] opted for a more sophisticated solution. Their semantics is given in terms of a constructive logic, and compilers perform a causality analysis [21, 22, 19, 23, 24, 20, 25] based on the computation of fix-points in a three-valued logic similar to Brzozowski and Seger's ternary simulation of asynchronous circuits [26]. 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. For instance, Berry and Mendler [27, 28] pointed out its equivalence to theorem proving in intuitionistic logic.

Causality cycles were first considered in hardware circuits (also called combinational cycles or feedback loops) in the early seventies [29, 30]: Kautz [29] and Rivest [30] proved that circuits with combinational cycles can be smaller than the smallest cycle-free circuits. For this reason, the introduction of combinational cycles has been recently proposed as a strategy for logic minimization [19, 20]. Furthermore, combinational cycles (called 'false paths' in this setting) occur in high-level synthesis of circuits by sharing common subexpressions [18].

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 coined the notion of constructive circuits [27, 28]. Due to the Curry-Howard isomorphism [31], the problem is also equivalent to type-checking functional programs. Finally, Edwards reformulates the problem such that the existence of dynamic schedules must be guaranteed for the execution of mutually dependent microsteps [23]. Hence, causality analysis is a fundamental algorithm that has already found many applications in computer science.

Malik [32] was first to present algorithms for the elimination of combinational cycles. Given input/output variables $\vec{x}$ and $\vec{y}$, the problem is to check whether a Boolean equation system $\vec{y} = \vec{\Phi}(\vec{x}, \vec{y})$ has a unique solution for all inputs. It is not difficult to see that this problem is NP-hard [32]. In order to obtain practically more efficient procedures, Malik used the embedding of Boolean algebra in ternary algebra as proposed by Bryant for the simulation of switch-level circuits [33].

The computation of a solution $\vec{y}$ depending on the inputs $\vec{x}$ is then reduced to the computation of a fix-point of the function $f_{\vec{x}}(\vec{y}) := \vec{\Phi}(\vec{x}, \vec{y})$, where the inputs $\vec{x}$ are fixed. The existence of such fix-points is guaranteed by the Tarski-Knaster theorem [34]. More details about causality analysis for synchronous programs may be found in [32, 11, 26, 22, 35, 21, 24, 36].

In general, causality checking is computationally expensive. Many optimizations have been reported in the literature making use of bounded model checking [37], classical logic [38], syntactic elimination and unrolling of cycles [19, 20, 23]. In the domain of polychronous programs, cyclic data dependencies were checked using SMT solvers in [39, 40].

Contribution. This article extends previous work [41] from which we exploit the proposed fixed-point theoretical framework to provide detailed proofs of constructiveness pertaining to determinism (Theorems 2 and 3). It also relates to earlier work on embedding Signal into Quartz [42] by providing an updated account to modeling Signal features using Quartz modules (Section 9), as well as its possible exploitation toward the implementation of Signal in a constructive framework.

Our work is rooted in a joint collaborative project, Polycore, in which we aim at combining the expressive capabilities of the imperative synchronous language Quartz and of 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) constructiveness 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 constructiveness and clock and causality analysis.

Our approach consists in the definition of a constructive framework for synchronous systems (Section 3) which encompasses the synchronous behavior of reactive modules (Section 4.1) and the multi-clocked behavior of polychronous data-flow networks (Section 5). This yields the very first executable semantics of SIGNAL that directly specifies an interpreter.

More importantly, this defines a constructive framework in which both synchronous modules and asynchronous networks can be represented (Section 6), allowing us to formulate formal properties of determinism for both synchronous modules, polychronous and asynchronous networks (Section 8).

This semantics is of interest on its own: it allows us to better understand the relationship between synchrony and polychrony, between constructiveness and endochrony, and to model causality as a formal verification problem. 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.

3. 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 implementation.

For this reason, one is usually interested in whether a program has a behavior that can be *constructively determined* by the operational semantics (a behavior that is unique for a given input or upon stabilization of delays). Such a *constructive* semantics is therefore based on fix-point iteration: it repeatedly executes iterations in order to infer the clocks and values of all signals at every logical instant, i.e., during every reaction.

3.1. Embedding Clocks and Values in a Complete Lattice

Fix-point theory is well understood in computer science, and theorems due to Kleene, Tarski, Knaster are often used to compute least and greatest fix-points [34, 43]. In order to make it easier to follow the rest of the paper, we define some of the essential concepts of fix-point theory.

Definition 1 (Complete Lattice). *A partially ordered set* $(\mathcal{D}, \sqsubseteq)$ *is a* lattice *if every pair* $\{x, y\} \subseteq \mathcal{D}$ *has a supremum* $\bigsqcup\{x, y\}$ *and an infimum* $\bigsqcap\{x, y\}$ *in*

$\mathcal{D}$. *$(\mathcal{D}, \sqsubseteq)$ is a* complete lattice *if, for* every *set $M \subseteq \mathcal{D}$, $\bigsqcup M$ and $\bigsqcap M$ exist in $\mathcal{D}$.*

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 $f(\bigsqcap M) = \bigsqcap f(M)$ hold for all (directed) sets $M \subseteq \mathcal{D}$ (every continuous function is monotonic).

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 $\mathcal{D}$ with the corresponding meaning as defined in Figure 1. Starting from $\mathcal{D}'$, we define a partial order $\sqsubseteq$ on $\mathcal{D}' \times \mathcal{D}'$: 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.

? signal status is unknown
$\bot$ signal is absent or inhibited
$\top$ signal is present or activated
$\lightning$ signal is inconsistent

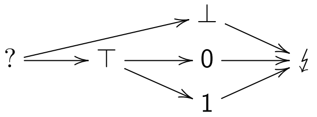

Figure 1: Embedding booleans into a complete lattice

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 following conditions:

1. The extension g' and the original operator g should yield the same results on $\mathcal{D}$ i.e. $\forall \vec{x} \in \mathcal{D}^n.\ g'(\vec{x}) = g(\vec{x})$.
2. The extension g' should be monotonic w.r.t. $\sqsubseteq$ (for finite $\mathcal{D}'$ this also implies continuity of g') i.e. $\forall \vec{x}, \vec{y} \in \mathcal{D}'^n.\ \vec{x} \sqsubseteq \vec{y} \Rightarrow g'(\vec{x}) \sqsubseteq g'(\vec{y})$.

These conditions predetermine most 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. $\sqsubseteq$ (similar to [36]).

Example. In Figure 2, we consider the extension of Boolean conjunction $\wedge$, defined on $\mathbb{B}^2 \to \mathbb{B}$. Its extension $\wedge'$ is defined 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 to $0 \wedge' 0 = 0$ 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 $1 \wedge' 1 = 1$. Hence, 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	$\lightning$
?	?	$\bot$	$\top$	0	$\top$	$\lightning$
$\lightning$	$\lightning$	$\lightning$	$\lightning$	$\lightning$	$\lightning$	$\lightning$

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

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

Figure 2: Embedding Boolean 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 boolean (or integer) operators in lattice $\mathcal{D}'$ have fix-points and, more interestingly, that they have uniquely defined least and greatest fix-points.

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: Our constructive semantics start with known input variables and some unknown input and local/output variables. If the least fix-point does no longer contain unknown values, the program has a unique behavior, it is constructive.

Theorem 1 (Fix-points in Lattices [34]). *Let $(\mathcal{D}, \sqsubseteq)$ be a complete lattice and $f : \mathcal{D} \to \mathcal{D}$ be a monotonic function. Then, f has fix-points and the set of fix-points even has a minimum and a maximum. If f is moreover continuous, then the least fix-point of f can be computed by the iteration $p_0 := \sqcap\mathcal{D}$, $p_{i+1} := f(p_i)$, and the greatest fix-point of f by $q_0 := \sqcup\mathcal{D}$, $q_{i+1} := f(q_i)$.*

4. Constructive Semantics of Quartz

The Quartz [2] language is a synchronous language that has been derived from the Esterel language [44, 27], but also from early prototype languages that were used at the University of Karlsruhe for formal verification like the synchronous hardware description language SHDL [45]. Similar to Esterel, the Quartz language is based on the paradigm of perfect synchrony [4, 5] which means that the execution of a program is divided into micro and macro steps.

The execution of micro steps is done in zero time, while the execution of a macro step requires one logical unit of time. All variables in Quartz have unique values for each macro step, and these values must be determined by the execution of the micro steps of that macro step. This is done by executing the micro steps in data-flow order, so that variables are not read before their value is determined in that macro step. As in Esterel, programs that do not allow such an execution are called non-constructive and are rejected by the operational semantics of the language [2]. As from the programmer's point of view, all macro steps take the same amount of logical time, concurrent threads run in lockstep and automatically synchronize at macro steps.

Variables in Quartz are statically typed and may have one of the types `bool` (boolean), `nat` (unsigned integers), `int` (signed integers), `bv` (bitvectors), `[N]`α (array of dimension `N` of type α), and tuples of any type, whereas the numeric and bitvector types can either be bounded or unbounded. In addition to the type, variable declarations also have to determine the storage mode which is currently either `event` (even variables) or `mem` (memorized variables). The storage mode determines the value of the variable in a step if no assignment of the program assigns a value to that variable. In that case, memorized variables maintain their previous value, while event variables are reset to a default value.

In contrast to Esterel and other synchronous languages, variables in Quartz are therefore always present, i.e., have a unique value at every point of time. This is due to the origin of the Quartz language in hardware design [45], since in synchronous hardware circuits variables (which denote the values of wires) do also have unique values at every clock tick.

Another consequence of the origin in hardware design is that Quartz distinguishes between immediate $x = \tau$ and delayed assignments $\texttt{next}(x) = \tau$ to variables: Both assignments evaluate the right-hand side expression τ in the current environment/macro step (where it is assumed that the variables that occur in τ are already known). Immediate assignments $x = \tau$ instantaneously transfer the obtained value of τ to the left-hand side x, whereas delayed ones $\texttt{next}(x) = \tau$ transfer this value in the following macro step. Immediate assignments therefore have to be scheduled according to their data-dependencies.

In contrast to delayed assignments, Esterel offers only immediate assignments and a `pre` operator to access the previous value of a variable. Moreover, values of variables in Esterel are determined by signal emissions $\texttt{emit x(}\tau\texttt{)}$ that make x present and determine its value as an immediate assignment in Quartz.

The remainder of this section gives a brief overview of the Quartz language and refers to [2] and to the Esterel primer [27] for its semantics. The core statements of the Quartz language consists of the following statements, provided that S, S_1, and S_2 are also core statements, ℓ is a location variable, x is a variable, σ is a Boolean expression, and α is a type:

- `nothing` (empty statement)
- $x = \tau$ and $\texttt{next}(x) = \tau$ (assignments)
- $\ell : \texttt{pause}$ (start/end of macro step)
- $\texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2$ (conditional)
- $S_1; S_2$ (sequential composition)
- $\texttt{do}\ S\ \texttt{while}(\sigma)$ (iteration)
- $S_1 \parallel S_2$ (synchronous concurrency)
- $\texttt{[weak] [immediate] abort}\ S\ \texttt{when}(\sigma)$
- $\texttt{[weak] [immediate] suspend}\ S\ \texttt{when}(\sigma)$
- $\{\alpha\ x;\ S\}$ (local variable x with type α and scope S)
- $\mathit{inst} : \mathit{name}(\tau_1, \ldots, \tau_n)$ (call of module *name*)

There is essentially only one basic statement that defines a control flow location, namely the `pause` statement. These statements are endowed with unique Boolean valued *control flow location variables* ℓ, which are true iff the control flow is currently at the statement $\ell : \texttt{pause}$. Since all other statements are executed in zero time (except for immediate suspensions), the control flow can only rest at these positions in the program.

In addition to the usual control flow constructs known from all kinds of imperative languages (conditionals, sequences and iterations), Quartz offers synchronous concurrency and preemption statements: The *parallel statement* $S_1 \parallel S_2$ immediately starts the statements S_1 and S_2. Then, both S_1 and S_2 run in lockstep, i. e. they automatically synchronize when they reach the following `pause` statement. The parallel statement runs as long as one of the sub-statements is active.

The meaning of *preemption statements* is as follows: A statement S which is enclosed by an `abort` block is immediately terminated when the given condition σ holds. Similarly, the suspend statement freezes the control flow in a statement S when σ holds. Thereby, two kinds of preemption must be distinguished: strong (default) and weak (indicated by keyword `weak`) preemption. While strong preemption deactivates both the control and data flow of the current step, weak preemption only deactivates the control flow, but retains the current data flow. The immediate variants check for preemption already at starting time, while the default is to check preemption only after starting time.

Modular design is supported first by the declaration of modules in the source code and second by calling modules in statements. Any statement can be encapsulated in a module, which further declares a set of input and output signals for interaction with its context statement. There are no restrictions for module calls, so that modules can be instantiated in every statement. Thus, in addition to parallel execution (which is only allowed in HDLs), a module instantiation can also be part of sequences or conditionals. Furthermore, it can be located in any abortion or suspension context, which possibly preempts its execution. Hence, a compiler that generates code for these modules is obviously challenged by this flexibility.

4.1. Constructive Synchronous Guarded Actions

We represent imperative Quartz modules using synchronous guarded actions [46, 47], as defined in Figure 3 and in the spirit of *guarded commands* [48, 49, 50, 51], 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 initial or immediate assignment $x = \tau$ writes the evaluated value of τ immediately to the variable x. A delayed assignment $\mathsf{next}(x) = \tau$ stores it and releases it onto the variable at the next execution step.

Immediate assignments define a causal dependency within the instant from all the read variables (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 value for the following instant. Moreover, there are initial actions $\mathsf{init}(x) = \tau$ without guards.

Guarded actions are composed by the operator $p\,|\,q$, and all actions defining a variable x are grouped by the operator $\mathsf{var}\,x : p\,\mathsf{default}\,v$. If none of these guarded actions on variable x apply, x is set to its *default value* v: Event variables $\mathcal{E} \subseteq \mathcal{V}$ are reset to their default value (like wires in hardware circuits), while memorized variables $\mathcal{M} \subseteq \mathcal{V}$ are reset to their previous value (like registers in hardware circuits). Finally, $\mathsf{done}\ p$ is used to mark guarded actions p as evaluated during the semantics.

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

Figure 3: Synchronous Guarded Actions

Notations. 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 $s \vdash \phi \rightharpoonup v$.

- For $\star \in \{\mathsf{and}, \mathsf{or}\}$, we write $s \vdash x \star y \rightharpoonup v$ iff $v = s(x) \star s(y) \in \mathbb{B}$,

- $s \vdash \mathsf{not}\,x \rightharpoonup v$ iff $v = \mathsf{not}\,s(x) \in \mathbb{B}$.

Notice that the relation $s \vdash \phi \rightharpoonup v$ 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. We call $dom(s)$ and $im(s)$ the domain and image of a store s and an update operation $s \uplus (x, v) = s \cup \{(x, s(x) \sqcup v)\}$ sets the status of x in s to the upper bound of its old $s(x)$ and new value v. Additionally, we write $V(p)$, $I(p)$ and $O(p)$ for the set of free, input and output signals of p.

Transition rules. 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 $s \uplus (x, v)$. If the guard evaluates to $\mathsf{0}$ then variable x remains unchanged and the action is marked as evaluated. In the case of a delayed action, the additional initial action $\mathsf{init}(x) = v$ is added to be applied in the following instant.

$$\frac{s \vdash \gamma \rightharpoonup \mathsf{1} \qquad s \vdash \tau \rightharpoonup v}{\begin{array}{c} s, \gamma \Rightarrow \mathsf{next}(x) = \tau \rightharpoonup s, \mathsf{done}\ (\mathsf{init}(x) = v\,|\,\gamma \Rightarrow \mathsf{next}(x) = \tau) \\ s, \gamma \Rightarrow x = \tau \rightharpoonup s \uplus (x, v), \mathsf{done}\ (\gamma \Rightarrow x = \tau) \end{array}}$$

$$\frac{s \vdash \gamma \rightharpoonup \mathsf{0}}{\begin{array}{c} s, \gamma \Rightarrow \mathsf{next}(x) = \tau \rightharpoonup s, \mathsf{done}\ (\gamma \Rightarrow \mathsf{next}(x) = \tau) \\ s, \gamma \Rightarrow x = \tau \rightharpoonup s, \mathsf{done}\ (\gamma \Rightarrow x = \tau) \end{array}}$$

Figure 4: Rules for immediate and delayed assignments

The rules for synchronous composition $p \mathbin{|} q$ (shown in Figure 5) additionally define the possible schedules of concurrent statements.

$$\frac{s, p \rightharpoonup s', p'}{s,\ q \mathbin{|} p \rightharpoonup s',\ q \mathbin{|} p'} \qquad \frac{s, p \rightharpoonup s', p'}{s,\ p \mathbin{|} q \rightharpoonup s',\ p' \mathbin{|} q}$$

Figure 5: Rules for synchronous composition

The done markers indicate the guarded actions that have already been executed in a step. They propagate by composition and blocks $\mathsf{var}\, x : p \,\mathsf{default}\, v$, Figure 6.

$$s,\ \mathsf{done}\ p \mathbin{|} \mathsf{done}\ q \rightharpoonup s, \mathsf{done}\ (p \mathbin{|} q) \qquad s,\ \mathsf{init}(x) = v \rightharpoonup s \uplus (x, v), \mathsf{done}\ ()$$

Once the body of a variable x is evaluated, the status of x is checked. If it is still unknown, it is set to the default value u stored with the variable block. The block is then marked as evaluated. If x has been assigned value v and x is an event variable, then again the block is just marked evaluated. If the variable is a memorized variable, however, its new value v is stored in place of u.

$$\begin{array}{l}
s \uplus (x, ?), \mathsf{var}\, x\colon \mathsf{done}\ (p)\ \mathsf{default}\, u \rightharpoonup s \uplus (x, u), \mathsf{done}\ (\mathsf{var}\, x\colon p \,\mathsf{default}\, u) \\
s \uplus (x, v), \mathsf{var}\, x\colon \mathsf{done}\ (p)\ \mathsf{default}\, u \rightharpoonup s \uplus (x, v), \mathsf{done}\ (\mathsf{var}\, x\colon p \,\mathsf{default}\, u),\ x \notin \mathcal{M} \\
s \uplus (x, v), \mathsf{var}\, x\colon \mathsf{done}\ (p)\ \mathsf{default}\, u \rightharpoonup s \uplus (x, v), \mathsf{done}\ (\mathsf{var}\, x\colon p \,\mathsf{default}\, v),\ x \in \mathcal{M}
\end{array}$$

Figure 6: Rules for synchronous blocks

Example. The above semantics of guarded actions is illustrated by the definition of a resettable counter. One first initializes the local counter x to 0. If the reset condition r is true, then next value of x is again reset to 0. If it is false, it is incremented by 1. The output o is set to the current value of the count x.

$$\mathsf{init}(x) = 0 \mathbin{|} o = x \mathbin{|} r \Rightarrow \mathsf{next}(x) = 0 \mathbin{|} \neg r \Rightarrow \mathsf{next}(x) = x + 1$$

Evaluation consists of scheduling the four equations in a causal order using the step relation $\rightharpoonup$ to progress on the status of all variables. Hence, we first set x to its current value 0, then the output o to x and then evaluate both actions guarded by the value of r to determine if x is next zeroed or incremented. We first set the input r to false:

$$\begin{array}{ll}
(r, \mathsf{0})(x, ?)(o, ?)\mathsf{init}(x) = 0 & \rightharpoonup (r, \mathsf{0})(x, 0)(o, ?)\mathsf{done}\,() \\
(r, \mathsf{0})(x, 0)(o, ?)o = x & \rightharpoonup (r, \mathsf{0})(x, 0)(o, 0)\mathsf{done}\,(x = o) \\
(r, \mathsf{0})(x, 0)(o, 0)r \Rightarrow \mathsf{next}(x) = 0 & \rightharpoonup (r, \mathsf{0})(x, 0)(o, 0)\mathsf{done}\,(r \Rightarrow \mathsf{next}(x) = 0) \\
(r, \mathsf{0})(x, 0)(o, 0)\neg r \Rightarrow \mathsf{next}(x) = x+1 & \rightharpoonup (r, \mathsf{0})(x, 0)(o, 0)\mathsf{done}\, r \Rightarrow \mathsf{next}(x) = x+1 \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathsf{init}(x) = 1
\end{array}$$

Starting from there, we set it to true:

$$\begin{array}{ll}
(r,\mathsf{1})(x,?)(o,?)\mathsf{init}(x) = 1 & \rightharpoonup (r,\mathsf{0})(x,1)(o,?)\mathsf{done}\,() \\
(r,\mathsf{1})(x,0)(o,?)o = x & \rightharpoonup (r,\mathsf{0})(x,1)(o,1)\mathsf{done}\,(x = o) \\
(r,\mathsf{1})(x,0)(o,0)\neg r \Rightarrow \mathsf{next}(x) = x+1 & \rightharpoonup (r,\mathsf{0})(x,1)(o,1)\mathsf{done}\,(r \Rightarrow \mathsf{next}(x) = x+1) \\
(r,\mathsf{1})(x,0)(o,0)r \Rightarrow \mathsf{next}(x) = 0 & \rightharpoonup (r,\mathsf{0})(x,1)(o,1)\mathsf{done}\, r \Rightarrow \mathsf{next}(x) = 0 \\
 & \qquad\qquad\qquad\qquad\qquad\quad \mathsf{init}(x) = 0
\end{array}$$

5. Polychronous Systems

In contrast to synchronous systems, polychronous specifications [52, 14] 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. The clock of a signal is encoded by its status which might be present or absent. The value of a signal can only be computed if its status is present.

Several frameworks have been considered to describe the formal semantics of SIGNAL: [53] proposes a logical framework in the form of a big-step deductive semantics, [54] a denotational, trace-theoretical framework in the form of tagged Mazurkiewicz traces, [55] a model of small-step synchronous automata, and [56] a game-theoretical micro-automata framework focusing on causality graphs. All these contributions relate to the present one, which provide the first, small-step, operational, executable semantics of SIGNAL, by using the simple trick of lifting its semantics domain from clocked values to the status.

5.1. Signal Specifications

In the remainder, we use SIGNAL to represent polychronous specifications. A SIGNAL process p consists of the composition $p\,|\,q$ of equations $x := y \star z$ built from primitive operators $\star \in \{\mathsf{and}, \mathsf{or}, \mathsf{not}, \$\mathsf{init}, \mathsf{when}, \mathsf{default}\}$. The construct p/x lexically binds the scope of a signal x to a process p.

$$p, q ::= x := y \star z \mid p\,|\,q \mid p/x \qquad \text{(process)}$$

Figure 7 gives the main transition rule for a ternary 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 outputs to a hypothetical triple (a, b, c). If so, a transition occurs and the status of signals (x, y, z) are updated.

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

Figure 7: Small-step operational semantics of equations

Let us first consider the case of a synchronous ternary equation $x := y\,\mathsf{and}\,z$ in Figure 8. From the initial status $s(x, y, z)$, there are three ways to progress: If one of the parameters is known to be absent, written $\perp$, then the other parameters must be deemed absent as well, as they all need to occur synchronously. A second case, shown in the middle of Figure 8, is when one parameter is known to be present. In that case, the other must be present as well. The last case, shown on the right of Figure 8, is when the values $a, b \in \mathbb{B}$ of the inputs are known. The only choice is to set the output to $a \wedge b$. If it is instead the output which is known to be true, and so must be the inputs.

$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{and}}$	$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{and}}$	$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{and}}$
$\perp$			$\perp\ \perp\ \perp$	$\top$			$\top\ \top\ \top$		a	b	$(a \wedge b)\ a\ b$
	$\perp$		$\perp\ \perp\ \perp$		$\top$		$\top\ \top\ \top$	1	a		$1\ a\ 1$
		$\perp$	$\perp\ \perp\ \perp$			$\top$	$\top\ \top\ \top$	1		a	$1\ 1\ b$

Figure 8: Transitions for conjunction

As a result of this transition table, one observes that information on absence or presence can possibly flow from the output of an equation back to its inputs, and possibly inhibits or trigger other signals in its context. The same principle applies to binary equations such as negation $x := \mathsf{not}\, y$. On the left of Figure 9, one of the parameters is absent. In that case, all parameters must be deemed absent. In the middle, one parameter is known to be present. Again, all parameters must be present. On the right, the value $a \in \mathbb{B}$ of a parameter is known and the other one must be set to its negation $\neg a$.

$s(x)$	$s(y)$	$\rightharpoonup_{\mathsf{not}}$	$s(x)$	$s(y)$	$\rightharpoonup_{\mathsf{not}}$	$s(x)$	$s(y)$	$\rightharpoonup_{\mathsf{not}}$
$\perp$		$\perp\ \perp$	$\top$		$\top\ \top$		a	$\neg a\ \ a$
	$\perp$	$\perp\ \perp$		$\top$	$\top\ \top$	a		$a\ \ \neg a$

Figure 9: Transitions for negation

Example. From the above rules, we can define an operational semantics for the clock synchronization operator in SIGNAL. The process $x\, \mathsf{sync}\, y$ only accepts signals x and y that have the same status, present with a value or absent, or yield an error otherwise:

$$x\, \mathsf{sync}\, y \stackrel{\Delta}{=} (a := (x = x) \,|\, b := (y = y) \,|\, c := (a = b))\, /abc$$

only accepts signals x and y that have the same status, present with a value or absent. Suppose that both x and y are present. Then, the equations $a := (x = x)$ and $b := (y = y)$ evaluate a and b to true. Therefore, $c := (a = b)$ must be true as well. Now, suppose that both x and y are absent. Then, the equations $a := (x = x)$ and $b := (y = y)$ must evaluate a and b to absent. Therefore, $c := (a = b)$ must be absent as well.

It remains to consider what happens if x is present and y is absent? In this case, $a := (x = x)$ makes a true and $b := (y = y)$ makes b absent. From the table above, we then have two choices to evaluate $c := (a = b)$, either propagate presence of a to b and c from $(\_, \top, \_) \rightharpoonup (\top, \top, \top)$, or absence of b to a and c from $(\_, \_, \bot) \rightharpoonup (\bot, \bot, \bot)$. In both cases, however, the rule (op) sums up that progress with the status of (a, b, c). Hence, assuming an initial status $s = (a, ?)(b, \bot)(c, \top)$, we have:

$$s,\, a := (b = c) \rightharpoonup s \uplus (a, \bot)(b, \bot)(c, \bot),\, a := (b = c)$$

This results in setting the status of c to $\bot \sqcup \top = \lightning$ and declaring the program as erroneous. In Signal, it is the task of a clock calculus [54] to statically check whether programs are free of synchronization errors (whose signal status can be deterministically computed from the program trigger).

Multi-clocked operators. The same evaluation principle can be applied to a downsampling equation $x := y \,\mathsf{when}\, z$ that possibly takes two signals y and z of different status or clock as arguments. The rules for propagating absence are the same. There is only one possible way to propagate presence, namely if the output value is known. If the input z is false, then the output x is deemed absent regardless of y. Finally, there is only one case where the output can have a value $a \in \mathbb{B}$, namely when z is true and y equals $a \in \mathcal{D}$.

$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{when}}$		
$\bot$			$\bot$	$\bot$	$\bot$
	$\bot$		$\bot$	$\bot$	$\bot$

$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{when}}$		
		$\bot$	$\bot$	$\bot$	$\bot$
		$\mathsf{0}$	$\bot$	$s(y)$	$\mathsf{0}$

$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{when}}$		
$\top$			$\top$	$\top$	$\mathsf{1}$
	a	$\mathsf{1}$	a	a	$\mathsf{1}$

$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{default}}$		
$\bot$			$\bot$	$\bot$	$\bot$
	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{default}}$		
	$\top$		$\top$	$\top$	$s(z)$
		$\top$	$\top$	$s(y)$	$\top$

$s(x)$	$s(y)$	$s(z)$	$\rightharpoonup_{\mathsf{default}}$		
	a		a	a	$s(z)$
	$\bot$	a	a	$\bot$	a

Figure 10: Transitions for sampling and merging

The 'default' operator $x := y \,\mathsf{default}\, z$ (also called a 'merge' operator) works opposite to sampling: its result x can only be ruled absent when both inputs are absent. On the contrary, there are many ways to propagate presence (it is sufficient that one argument is present): If y is present, then its value is forwarded to x, otherwise if z is present, then its value is forwarded to x. If both are absent, so is x.

Delay equation. The case of the delay equation $x := y \,\mathsf{\$init}\, a$ requires a specific rule (delay) to take care of the fact that its third argument, the state, is a value $a, d \in \mathcal{D}$ (which we could have avoided by making locations explicit).

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

Figure 11: Operational semantics of delay

Apart from this peculiarity, delay behaves just like a synchronous operator:

$s(x)$	$s(y)$	$\rightharpoonup_{\mathsf{\$init}\,a}$	$s(x)$	$s(y)$	$\rightharpoonup_{\mathsf{\$init}\,a}$	$s(x)$	$s(y)$	$\rightharpoonup_{\mathsf{\$init}\,a}$
	$\bot$	$\bot\ \bot\ \ a$		$\top$	$a\ \top\ \ a$		b	$a\ b\ \ b$
$\bot$		$\bot\ \bot\ \ a$	$\top$		$a\ \top\ \ a$			

Restriction. The composition of equations $p\,\|\,q$ behaves as in Quartz by choosing a schedule of transitions among the composed equations of p and q. Restriction p/x lexically binds the scope of a local signal x to a given process p.

$$\frac{s \uplus (x, ?), p \rightharpoonup^{*} s' \uplus (x, a), p'}{s,\ p/x \rightharpoonup s',\ p'/x} \quad (x \notin dom(s) \cup dom(s') \wedge a \in \mathcal{D}^{\bot})$$

Figure 12: Rules for restriction

Its goal, Figure 12, is to determine the status a of the signal x which must be either absent or present with a value ($a \in \mathcal{D}^{\bot} = \mathcal{D} \cup \{\bot\}$) from the initial unknown status. This amounts to choosing a series $\rightharpoonup^{*}$ of transitions among equations in p.

Triggering execution. None of the rules presented so far is able to handle the case when all signals of a process have an unknown status. They are deductive rules, allowing one to infer the status implied by an individual equation from knowledge accumulated from its context. When execution starts, however, the initial status s of all signals x of a process p are unknown: $im(s) = \{?\}$.

Execution can only start by using a trigger rule, shown in Figure 13, which either inhibits some signal x to let a process p stutter (i.e. $s \uplus (x, \bot)$), or triggers the execution of p by instead setting it to present (i.e. $s \uplus (x, \top)$) and lets the deductive rules apply on p to determine the status of all other signals in s.

$$s, p \rightharpoonup s \uplus (x, \bot), p \quad s, p \rightharpoonup s \uplus (x, \top), p \qquad (x \in dom(s) \wedge im(s) = \{?\})$$

Figure 13: Trigger rules

To the very end of choosing a signal x to determine the status of all other signals, it is necessary to consider the synchronization relations implied by the equations in p and to use the clock calculus of Signal [54] in order to determine a nonempty subset $T_p \subseteq V(p)$ of signals whose status imply that of all the other ones. This, in turn, amounts to choosing a strategy to schedule the execution of the program or, in game-theoretical terms, to find a winning strategy against the program.

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. If that count reaches 0, the counter synchronizes with the input signal n to reset the count. The counter has input n and output o. The local signal c is the current count, x its decrement, and y the reset condition.

$$count_0(n,o) \triangleq \left(\begin{array}{lrl} & c := & o\,\mathsf{\$init}\,0 \\ | & o := & n\,\mathsf{default}\,x \\ | & x := & c-1 \\ | & y := & 1\,\mathsf{when}\,(c=0)) \\ | & & n\,\mathsf{sync}\,y \end{array}\right)/cxy$$

The execution of the counter is depicted by a series of steps. Changes are marked with a bullet $\bullet$. The internal state of the counter is underscored $count_0$. Let us assume for now that the environment has triggered execution of the program by furnishing the input signal n with the value 1 to start counting. This allows us to determine the output o of the counter by the merge rule. Consequently, and from the delay rule, the initial count 0 can be loaded into c and the new 1 is stored in place of it. Once c is known, x can be decremented by the subtraction rule, and y be deduced absent by the sampling rule. We are left with the synchronization constraint $n\,\mathsf{sync}\,y$ which, luckily, is true, as both n and y are present.

$$\begin{array}{llllllll}
\multicolumn{8}{l}{(c,?)(n,1_\bullet)(o,?)(x,?)(y,?),\ count_0} \\
\rightharpoonup (c,?) & (n,1) & (o,1_\bullet) & (x,?) & (y,?) & count_0 & (\text{from } \rightharpoonup_{\mathsf{default}}) \\
\rightharpoonup (c,0_\bullet) & (n,1) & (o,1) & (x,?) & (y,?) & count_{1_\bullet} & (\text{from } \rightharpoonup_{\mathsf{\$init}}) \\
\rightharpoonup (c,0) & (n,1) & (o,1) & (x,-1_\bullet) & (y,?) & count_1 & (\text{from } \rightharpoonup_{\mathsf{sub}}) \\
\rightharpoonup (c,0) & (n,1) & (o,1) & (x,-1) & (y,1_\bullet) & count_1 & (\text{from } \rightharpoonup_{\mathsf{when}}) \\
\rightharpoonup (c,0) & (n) & (o,1) & (x,-1) & (y,1) & count_1 & (\text{from } \rightharpoonup_{\mathsf{sync}})
\end{array}$$

Let us continue with a second round of execution. Only now, the environment does not deliver a new count and the status of all the signals is unknown. We therefore have to first use the trigger rule to start execution. Here, the aim of the game is to do so in a way that allows all equations to execute. To find a winning strategy, we can use the synchronization relations implied by the counter to determine which signal to trigger.

From the delay, subtraction and merge equations, we can first deduce that c, o, and x are synchronous. We assign them to an equivalence class $V_1 = \{c,o,x\}$ which means that, as soon as one of them is known to be present or absent, all the others have the same status. Next, there is an explicit synchronization constraint between n and y, hence let $V_2 = \{n,y\}$. Now, since n is defined by a sampling of c, we can deduce n from c, hence:

$$\text{status of } V_1 = \{c,o,x\} \text{ known} \Rightarrow \text{status of } V_2 = \{n,y\} \text{ known}$$

Therefore, if we set either of c, o or x present, the program should be able to determine the status of all other signals. Since the value of o depends on x

which depends on c, we choose to set the count c present. From the rule of delay, this has the effect of loading the current state 1 of the count into c and setting the output o present.

We can now execute both the subtraction, setting x to 0, and the sampling, setting y to absent. Once we know the status of y, we can set the status of n to absent as well by the synchronization rule. The output o can now be determined to be x by the merge rule and, concurrently, its value can be stored as the new state of the counter due to the delay equation.

$$\begin{array}{l}
(c,?)(n,?)(o,?)(x,?)(y,?),\ \mathit{count}_1 \\
\quad \rightharpoonup (c,\top_\bullet)(n,?)\ (o,?)\ (x,?)\ (y,?)\ \mathit{count}_1 \\
\quad \rightharpoonup (c,1_\bullet)\ (n,?)\ (o,\top_\bullet)(x,?)\ (y,?)\ \mathit{count}_1\ \text{(from } \rightharpoonup_{\mathsf{\$init}}) \\
\quad \rightharpoonup (c,1)\ (n,?)\ (o,\top)\ (x,0_\bullet)(y,?)\ \mathit{count}_1\ \text{(from } \rightharpoonup_{\mathsf{sub}}) \\
\quad \rightharpoonup (c,1)\ (n,?)\ (o,\top)\ (x,?)\ (y,\bot_\bullet)\mathit{count}_1\ \text{(from } \rightharpoonup_{\mathsf{when}}) \\
\quad \rightharpoonup (c,1)\ (n,\bot_\bullet)(o,\top)\ (x,0)\ (y,\bot)\ \mathit{count}_1\ \text{(from } \rightharpoonup_{\mathsf{sync}}) \\
\quad \rightharpoonup (c,1)\ (n,\bot)\ (o,0_\bullet)\ (x,0)\ (y,\bot)\ \mathit{count}_1\ \text{(from } \rightharpoonup_{\mathsf{default}}) \\
\quad \rightharpoonup (c,1)\ (n,\bot)\ (o,0)\ (x,0)\ (y,\bot)\ \mathit{count}_{0\bullet}\ \text{(from } \rightharpoonup_{\mathsf{\$init}})
\end{array}$$

As the example shows, we obtain a constructive and executable operational semantics that encompasses the behavior of polychronous Signal processes. Oversampling, a distinctive expressive capability of Signal, can be easily modeled in our operational semantics of polychronous processes. For instance, we can embed the counter into the computation of a sum:

$$\mathit{sigma}(i,j) \stackrel{\Delta}{=} (\mathit{count}(n,o)\,|\,k := (k\,\mathsf{\$init}\,0) + o\,|\,j := k\,\mathsf{when}\,(o=0))\,/kno$$

The only difference between the processes *count* and *sigma* is that execution of *count* is triggered from its signal o while *sigma* is triggered internally by the signal k. Both o and k can be statically determined as the triggers of these programs by the clock relations implied by the equations and by the clock calculus of Signal [54]. Yet, the processes *count* and *sigma* have different formal properties, as will be mentioned later.

6. 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 (finite) stream by an inductive sequence $w \in \mathbb{S} = \mathcal{D}^*$ of values a.

We define the operation of reading the oldest value a from a stream as $a.w$ and writing the newest value a onto it as $w.a$ in order to reflect a first-in-first-out protocol. The empty FIFO is written ϵ. 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.

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

Then, we simply lift the transition relation $s, p \rightharpoonup t, q$ to further account for the way a process p interacts with its 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.

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

Now, the way Quartz modules and Signal processes interface with the environment differs. Whereas a module is reactive and starts execution once all its inputs are available, a process is said to be endochronous and controls its inputs once it receives control from a trigger.

Interfacing modules and processes. Hence, in Figure 14, a value a will be loaded to an input signal x of a process p only if it is triggered present. The output of a process onto streams is performed when computations within the process are completed. It is defined by the 'flush' relation $E, \langle s, p\rangle \rightharpoondown F, \langle s, p\rangle$, which pulls computed output values from the local store to the shared streams. The application of this relation delimits the temporal barrier of a synchronous instant or reaction. It also filters inputs which have been read and signals that are absent. No rule applies to a signal whose status is in error.

$$\begin{array}{rll}
E \uplus (x, a.w), \langle s(x, \top), p\rangle \rightharpoonup E \uplus (x, w), \langle s(x, a), p\rangle & & (x \in I(p)) \\
E \uplus (x, w), \langle s \uplus (x, a), p\rangle \rightharpoondown E \uplus (x, w.a), \langle s \uplus (x, ?), p\rangle & & (x \in O(p) \wedge a \in \mathcal{D}) \\
E, \langle s \uplus (x, a), p\rangle \rightharpoondown E, \langle s \uplus (x, ?), p\rangle & (a = \bot) \vee (x \in I(p) \wedge a \in \mathcal{D}) &
\end{array}$$

Figure 14: Interface semantics of polychronous equations p

The asynchronous interface of a synchronous Quartz module is simpler. It only requires rules for the flush relation $\rightharpoondown$ to write the computed outputs and to read the next values of inputs.

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

Figure 15: Interface semantics of synchronous modules p

The constructive GALS semantics $E, P \to E, Q$ can be defined for synchronous modules and polychronous processes alike. It consists of local synchronous execution steps $E, P \rightharpoonup E, Q$ during which inputs are read and outputs are computed, and a synchronization step $E, P \rightharpoondown^* F, Q$, during which outputs are flushed. Scheduling the execution of parallel composition $P \parallel Q$ is defined by interleaving the execution steps of P and Q.

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

Figure 16: Interface semantics of synchronous modules p

Communication from a process to another one is assumed to be point-to-point. Broadcast communication can be simulated by multiplexing the output of the writer to multiple readers: for any $x \in \mathit{dom}(E)$ in the domain of E and such that $x \in O(P)$ with $x \in I(Q)$ and $x \in I(R)$, one can rewrite $P \parallel Q \parallel R$ as

$$P \parallel (Q[y/x] \parallel R[z/x] \parallel \langle (), y := x \,|\, z := x\rangle) /yz$$

for any y, z not in Q, R. Multiple writers on a single stream are not allowed as per the synchronous paradigm.

7. Determinism, reactivity and endochrony

The layered constructive semantics allows us to formulate classical properties of polychronous processes, as stated in the trace or logical settings of [54], yet in a constructive operational semantics framework. We first state properties pertaining to the correctness of QUARTZ programs. A synchronous QUARTZ module p is deterministic iff for any combination of input values, its transition function always terminates by producing a combination of output values.

Synchronous determinism relates to the property of reactivity in ESTEREL. However, while reactivity in ESTEREL assumes that all input status of a module are known (present or absent), determinism in QUARTZ instead assumes that the input values of a synchronous QUARTZ module are known (or synchronous). If one of the input is absent, it will invariably stutter.

Definition 2 (Synchronously deterministic module). *A module p is deterministic iff for every input status $s = s_p^? \uplus_{x \in I(p)}^{a \in \mathcal{D}} (x, a)$, there exists a unique t defined on $\mathcal{D}$ such that $s, p \rightharpoonup^* t, q$.*

Example. Recall the resettable counter we defined using guarded actions. It has input r, output o and a local variable x. One first initializes the local counter x to 0. If the reset condition r is true, then the next value of x is set to 0. If it is false, it is incremented by 1 and the next value of x is stored. Hence, for all values $a \in \mathbb{B}$ of the input r, there exists a unique value x of the output o: the counter is synchronously deterministic.

$$\mathsf{init}(x) = 0 \,|\, o = x \,|\, r \Rightarrow \mathsf{next}(x) = 0 \,|\, \neg r \Rightarrow \mathsf{next}(x) = x + 1$$

Synchronous determinism applied to the case of SIGNAL processes takes into account the case that some inputs may not always be read (like the input signal n of the counter). Therefore, we say that a polychronous SIGNAL process p is deterministic iff for an initially unknown status and a triggered signal, its step relation always converges to a unique fix-point. Clearly, this assumption needs to be embedded in an environment that provides input value streams E.

As a result, determinism relates to endochrony [54], the capability of a process to internally choose which input streams to read and which output streams to write at all times. Let us note $s_p^? = \{(x, ?) \mid x \in V(p)\}$ for the initial store of a process p, we say that a polychronous process p is deterministic iff for any input stream E, it yields a unique output stream F and final state q when triggered from either of a given set of signals $T_p \subseteq V(p)$, its triggers.

Definition 3 (Polychronous determinism). *A process p is asynchronously deterministic iff there exists $\emptyset \subset T_p \subseteq V(p)$ such that for any environment E defined on $V(p)$ and any $x \in T_p$, there exists a unique (F, q) such that $E, \langle s_p^? \uplus (x, \top), p\rangle \rightharpoonup^* \rightarrow^* F, \langle s_p^?, q\rangle$.*

Example. Recall the polychronous counter of the previous section. Every time its output o is triggered, it provides a count, its decrement and calculates its reset condition y. If true, the input signal n is triggered, read and fed to the output o. Otherwise, the current count c is considered. As a result, for $T_{count} = \{o\}$, and for any input trace E defined on $\{n, o\}$, there exists a unique output trace F that can be deterministically computed.

$$
\begin{aligned}
&\mathit{count}_0(n, o) \triangleq \\
&\quad (c := o \,\mathsf{\$init}\, 0 \mid o := n \,\mathsf{default}\, x \mid x := c - 1 \mid y := 1 \,\mathsf{when}\, (c = 0)) \mid n \,\mathsf{sync}\, y) \,/cxy
\end{aligned}
$$

Endochrony (or asynchronous determinism) is, by definition, a trace property and must therefore be defined by a fixed-point of the big-step relation $\rightarrow$ in order to make a trace explicit. For convenience, an inductive trace can be represented by an environment E whose streams are of inductive size. As a result, checking endochrony is, in general, undecidable. Nonetheless, it is implied by asynchronous determinism.

Definition 4 (Endochrony). *A process p is endochronous iff for any input trace E, there exists a unique output trace F such that $E, p \rightarrow^* F, q$.*

Example. Process *count* is an example of endochronous process. If one repeatedly executes it with some input trace, for instance $E = \{(n, 1.2.3), (o, \epsilon)\}$, the output trace is invariably $F = \{(n, \epsilon), (o, 0.1.0.2.1.0.3.2.1.0)\}$.

Lemma 1 (Polychronous determinism implies endochrony). *If a module p is asynchronously deterministic, then it is endochronous.*

PROOF. By hypothesis, process p is deterministic. By Definition 3, a process p is asynchronously deterministic iff there exists $\emptyset \subset T_p \subseteq V(p)$ such that for any environment E defined on $V(p)$ and any $x \in T_p$, there exists a unique (F, q) such that $E, \langle s_p^? \uplus (x, \top), p\rangle \rightharpoonup^* \rightharpoondown^* F, \langle s_p^?, q\rangle$. By definition of the step relation $\rightarrow$, $E, p \rightarrow F, q$ always yields a unique F. Therefore, by induction on the iteration $E, p \rightarrow^* F, q$ from p and E, the fixed-point F is unique. By Definition 4, p is endochronous. □

Lemma 2 (Synchronous determinism implies endochrony). *If a module p is synchronously deterministic, then it is also endochronous.*

Proof. By hypothesis, consider a synchronously deterministic Quartz module p. From Definition 2, all input valuations $s = \{(x, a) \,|\, x \in I(p), a \in \mathcal{D}\} \cup \{(y, ?) \,|\, y \in V(p) \setminus I(p)\}$ yield a unique s defined on $\mathcal{D}$ such that $s, p \rightharpoonup^* t, q$. Now, consider an environment E defined on $V(p)$. If one of the input streams of E is empty, then $\rightarrow^* = \rightarrow^0$. The interface semantics $\rightharpoondown$ for Quartz module defines a unique rule to read input values from input streams $E \uplus (x, a.w), \langle s, p\rangle \rightharpoondown E \uplus (x, w), \langle s_x \uplus (x, a), p\rangle$ and another unique rule to write output values onto output streams $E \uplus (x, w), \langle s \uplus (x, a), p\rangle \rightharpoondown E \uplus (x, w.a), \langle s_x \uplus (x, ?), p\rangle$. Therefore, $E, \langle s_p^?, p\rangle \rightharpoondown^* F, \langle s, p\rangle \rightharpoonup^* F, \langle t, q\rangle \rightharpoondown^* G, \langle u, r\rangle$ yields a unique environment G, store u and state r. The store u differs from $s_p^?$ with all input values loaded from E and F. From Definition 4, the module p is endochronous. □

Example. Exercising repeated execution of the resettable counter with the input trace $\{(n, 1.0.1.0.0.1.0.0.0.1), (o, \epsilon)\}$ yields only one possible output trace $\{(n, \epsilon), (o, 0.1.0.1.2.0.1.2.3.0)\}$.

A Signal process should finally be invariant to stuttering to enjoy the criteria of weak-endochrony [57], i.e., a process of inhibited triggers should not react to absence.

Definition 5 (Stuttering). *A process p is stuttering-robust iff for any $x \in T_p$, $\langle s_p^? \uplus (x, \bot), p\rangle \rightharpoonup^* \langle s, p\rangle$ and $im(s) = \{\bot\}$.*

Example. The *count* process is stuttering-robust because, if its output o is set to absent, then the status of all other signals become absent. This is not the case of process *sum*.

8. Synchronous and asynchronous constructiveness

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 fix-point theoretic implication of its definition on that continuous domain and try to formulate the property of constructiveness. Originally, constructiveness was defined as a property of an imperative synchronous module that pertains to the reachability of a stable state in its electrical semantics [22]. In the operational setting of the Quartz language, this means that given any combination of input values, a synchronous module p should always define its output values.

Definition 6 (Synchronous constructiveness). *A module p is constructive iff the fix-point $s, p \rightharpoonup^* t, q$ of all derivation from an input valuation $s = s^?_p \uplus^{a \in \mathcal{D}}_{x \in I(p)} (x, a)$ yields a store t defined on $\mathcal{D}$.*

Thanks to the fidelity level of our small-step operational framework, the formulation of constructiveness matches that of determinism stated in Section 7.

Theorem 2 (Constructiveness and determinism). *If a module p is constructive, then it is synchronously deterministic.*

Proof. By hypothesis, let $p = p_0$ be a synchronous module and consider any initial valuation s_0 of its input variables. By Definition 2, there exists an evaluation sequence $s_0, p_0 \rightharpoonup^* s_n, p_n$ for some $n \geq 0$ which yields a store s_n whose variables are all defined in $\mathcal{D}$:

$$(1) \quad \left(\forall s_0 \left| \begin{array}{l} s_0(x) \in \mathcal{D}, x \in I(p_0) \\ s_0(x) = ?,\ x \notin I(p_0) \end{array} \right.\right) (\exists (s_i, p_i)_{0 \leq i \leq n} \in \rightharpoonup^*) \, (\forall x \in V(p_n)) \, s(x) \in \mathcal{D}$$

From (1), the pair (s_n, p_n) is a fix-point obtained by applying the transitive closure of the step relation $\rightharpoonup$. By induction, the step relation is monotonic and continuous on $\mathcal{D}'$:

- The small step relation $\rightharpoonup_\star$ of all operators $\star \in \{\mathsf{and}, \mathsf{or}, \mathsf{not}\}$ is defined by a monotonic and continuous functions on $\mathcal{D}'$ (i.e. completed with $\lightning$). Therefore, the step relation $s,\ x := y \star z \rightharpoonup t,\ x := y \star z \star$ is monotonic and continuous for all base operators.

- By induction hypothesis, if $s, p \rightharpoonup t, q$ is monotonic and continuous, then so are $s, p \,\|\, r \rightharpoonup t, q \,\|\, r$ and $s, r \,\|\, p \rightharpoonup t, r \,\|\, q$, hence, also the step relation relation of composition.

As a consequence, the pair (s_n, p_n) is a fix-point of a monotonic and continuous iterative application of the transition relation $\rightharpoonup$. By application of the fix-point Theorem 1, it is therefore the unique and least fixed-point of $\rightharpoonup^*$. By Definition 2, this implies that the module p is synchronously deterministic. □

We shall now formulate constructiveness in the context of polychronous processes. However, it is clear from the example of the counter that a polychronous process is not necessarily constructive in the sense of Quartz: it may not be reactive. Unlike a Quartz module, the counter process triggers a sequence of execution steps every time its trigger o is activated. It only loads its input count n when its internal count c reaches 0. Hence, it is not reactive w. r. t. its input signals, but it is endochronous w. r. t. its input streams.

$$\begin{array}{l} \mathit{count}_0(n, o) \overset{\Delta}{=} \\ \quad (c := o \,\$\mathsf{init}\, 0 \,\|\, o := n \,\mathsf{default}\, x \,\|\, x := c - 1 \,\|\, y := \mathsf{1}\,\mathsf{when}\,(c = 0)) \,\|\, n \,\mathsf{sync}\, y) \,/cxy \end{array}$$

This observation yields a more general (asynchronous) formulation of constructiveness: a process p is asynchronously constructive iff for any combination of values available from its input streams, it always produces an output.

Definition 7 (Polychronous constructiveness). *A process p is constructive iff for any $x \in T_p$ and environment E, the fix-point of the derivation $E, \langle s_p^? \uplus (x, \top), p\rangle \rightharpoonup^* F, \langle s, q\rangle$ yields a unique status s defined on $\mathcal{D}^\perp$.*

We claim that the transitive closure $\rightharpoonup^* F$ of a constructive process always yields a unique valuation defined on $\mathcal{D}^\perp$. If it isn't, 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 $\lightning$. Asynchronous constructiveness corresponds to the property of asynchronous determinism in the denotational framework of Signal [54, 57, 58].

Theorem 3 (Constructiveness and endochrony). *Every constructive process p is deterministic.*

Proof. By hypothesis, consider an environment $E_0 = E$, a process $p = p_0$ and an initially empty valuation $s_0 = s_p^? \uplus (x, \top)$ with $x \in T_p$. By Definition 3, there exists an evaluation sequence $E_0, s_0, p_0 \rightharpoonup^* E_n, s_n, p_n$ for some $n \geq 0$ such that s_n is defined on $\mathcal{D}^\perp$:

$$
\begin{array}{lll}
(2) & (\forall E_0 \,|\, dom(E_0) = V(p_0)\,) & \\
 & (\exists (E_i, s_i, p_i)_{0 \leq i \leq n} \in \rightharpoonup^*) & \\
 & (\forall x \in V(p_n)) & s_n(x) \in \mathcal{D}^\perp
\end{array}
$$

From (2), the triple (E_n, s_n, p_n) is a fix-point obtained by applying the transitive closure of the step relation $\rightharpoonup$. By using the same induction principle as previously, we show that the step relation is monotonic and continuous on $\mathcal{D}'$:

- The small step relation $\rightharpoonup_\star$ of all polychronous operators is defined by a monotonic and continuous functions on $\mathcal{D}'$ (i.e., completed with $\lightning$). Therefore, the step relation $s,\ x := y \star z \rightharpoonup t,\ x := y \star z$ is monotonic and continuous for all base operators.

- The extension of the step relation $s, p \rightharpoonup t, q$ to triples $E, s, p \rightharpoonup t, q, F$ has one rule: for any $x \in I(p)$, the transition $E \uplus (x, a.w), \langle s \uplus (x, \top), p\rangle \rightharpoonup E \uplus (x, w), \langle s \uplus (x, a), p\rangle$ strictly decreases the input stream associated to x in E and updates the status of x from present to value a. Hence, $E, s, p \rightharpoonup t, q, F$ is monotonic and continuous.

- By induction hypothesis, if $E, s, p \rightharpoonup F, t, q$ is monotonic and continuous, then so are $E, s, p \,|\, r \rightharpoonup F, t, q \,|\, r$ and $E, s, r \,|\, p \rightharpoonup F, t, r \,|\, q$, hence the step relation relation of composition is also monotonic and continuous.

As a consequence, the triple (E_n, s_n, p_n) is the fix-point of a monotonic and continuous iterative application of the relation $\rightharpoonup$. By application of the fix-point theorem 1, it is the unique least fixed-point of $\rightharpoonup^*$. By Definition 3, this implies that the process p is polychronously deterministic. $\Box$

9. Discussions

Towards compositional desynchronisation. Definition 7 of polychronous constructiveness is formulated in a way that coincides with a class of systems which can be deterministically executed starting from a singleton trigger. This indeed characterizes so-called endochronous systems in the sense of [54] whose input/output signal status can all be decided from that of a single one called '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 [57, 58] (where $E.F$ denotes the concatenation of streams):

Definition 8 (Weak endochrony). *A process p is weakly-endochronous if it is stuttering-robust and, for any*

- *environment E_1, E_2 s.t. $T_p = T_1 \uplus T_2$, $T_1 = dom(E_1)$ and $T_2 = dom(E_2)$,*
- *status $s_1 = s_p^? \uplus \{(x, \top) \,|\, x \in T_1\}$ and $s_2 = s_p^? \uplus \{(x, \top) \,|\, x \in T_2\}$,*

it is the case that

- *if $E_1.E, \langle s_1, p\rangle \rightharpoonup^* F_1, \langle t_1, p_1\rangle$ and $E_2.E, \langle s_2, p\rangle \rightharpoonup^* F_2, \langle t_2, p_2\rangle$*
- *then $E_2.F_1, \langle s_2, p_1\rangle \rightharpoonup^* F, \langle u_1, q_1\rangle$ and $E_1.F_2, \langle s_1, p_2\rangle \rightharpoonup^* G, \langle u_2, q_2\rangle$*

yield u_1, u_2 defined on $\mathcal{D}^\perp$, equivalent q_1 and q_2, and equivalent F and G.

Weakly-endochronous systems are known to be compositional, i.e., weak endochrony is preserved by composition. To allow this, however, one needs to allow some triggers to possibly be absent during a given execution step. All computations depending of that trigger would then need to evaluate to absent. This can be done by (internally) choosing to execute the inhibition rule instead of the trigger. In our framework, a (weakly) endochronous process p is characterized by a set of independent triggers T_p. Each of them triggers execution of independent computations. Such a process should of course be robust to any possible schedule that one may choose between these independent triggers and yield a confluent state i. e. it should be constructive and continuous.

Definition 9 (Weak constructiveness). *A process p is weakly constructive if for any environment E and store $s = s_p^? \uplus \{(x, a) \,|\, x \in T(p), a \in \{\perp, \top\}$, the derivation $E, \langle s, p\rangle \rightharpoonup^* F, \langle t, q\rangle$ yields t defined on $\mathcal{D}^\perp$.*

We claim that by checking constructiveness for all possible partitions of the triggers of p, we can check that p is weakly endochronous. The detailed proof of this property does in fact mostly reuse the techniques used for Theorem 3.

Proposition 1 (Weak constructiveness and endochrony).
A weakly constructive process is weakly endochronous

Example. The following example illustrates the concepts of weak constructivity. It consists of two similar interdependent lines, which compute some partial results s_1 and s_2. If the mode of the first equation is set to $m_1 = 1$, it can operate independently of the second one. The same applies to the second one for mode $m_2 = 2$. If the modes of both equations are set to 3, the partial result s_2 of the second equation is used in the first one, and if the modes are set to 4, the information flows in the opposite direction.

$$
\mathit{Constr}(a_1, a_2, b_1, b_2, m_1, m_2, s_1, s_2) \triangleq
\left(\begin{array}{l}
\quad s_1 := (a_1 \times b_1 - c_1)\,\mathsf{when}\,(m_1 \neq 2)\\
\mid c_1 := (s_2\,\mathsf{when}(m_1 \neq 4))\,\mathsf{default}\,0\\
\mid s_2 := (a_2 \times b_2 - c_2)\,\mathsf{when}\,(m_2 \neq 1)\\
\mid c_2 := (s_1\,\mathsf{when}(m_2 \neq 3))\,\mathsf{default}\,0\\
\mid \qquad (\mathsf{when}\,m_1 = 3)\,\mathsf{sync}\,(\mathsf{when}\,m_2 = 3)\\
\mid \qquad (\mathsf{when}\,m_1 = 4)\,\mathsf{sync}\,(\mathsf{when}\,m_2 = 4)
\end{array}\right)/c_1c_2
$$

This Signal specification is weakly constructive. The first line and the second line can be executed independently, there are several triggers so that different schedules are possible. However, all of them lead to the same asychronous input/output behavior. Another interesting aspect is the cyclic dependency between s_1 and s_2 when both parts operate simultaneously. This cycle is dynamically broken by our contructive framework either at c_1 (by $m_1 = m_2 = 3$) or at c_2 (by $m_1 = m_2 = 4$).

Co-modeling with Signal *and* Quartz. As we used a common operational framework for the definition of constructiveness in Signal and Quartz, we can handle specifications combining parts of both worlds. For instance, consider the following example, which has a similar functionality than the previous one. However, the computations of s_1 and s_2 are now replaced by synchronous guarded actions.

$$
\mathit{Mix}(a_1, a_2, b_1, b_2, m_1, m_2, s_1, s_2) \triangleq
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathsf{var}\,s_1: \begin{array}[t]{ll} \quad m_1 = 3 & \Rightarrow s_1 = a_1 \times b_1 - c_1\\ \mid m_1 \mathsf{mod}\,2 = 1 & \Rightarrow s_1 = a_1 \times b_1 - c_1 \end{array}\\
\mathsf{default}\,0\\
\mid \mathsf{var}\,s_2: \begin{array}[t]{ll} \quad m_2 = 4 & \Rightarrow s_2 = a_2 \times b_2 - c_2\\ \mid m_2 \mathsf{mod}\,2 = 0 & \Rightarrow s_2 = a_2 \times b_2 - c_2 \end{array}\\
\mathsf{default}\,0
\end{array}\right)\\
\mid \left(\begin{array}{l}
\quad c_1 := s_2\,\mathsf{default}\,0\\
\mid c_2 := s_1\,\mathsf{default}\,0\\
\mid \qquad (\mathsf{when}\,m_1 = 3)\,\mathsf{sync}\,(\mathsf{when}\,m_2 = 3)\\
\mid \qquad (\mathsf{when}\,m_1 = 4)\,\mathsf{sync}\,(\mathsf{when}\,m_2 = 4)
\end{array}\right)
\end{array}\right)/c_1c_2
$$

We have a similar cyclic dependency between s_1 and s_2 as in the previous example. This time, it is broken in the synchronous guarded actions. If

$m_1 = m_2 = 3$, both guarded actions of the second line cannot fire, which gives s_2 the default value and thereby, makes it independent of s_1. Thus, the cycle in the synchronous guarded actions are not broken by abent values as in the previous Signal example. Nevertheless, both variants are covered by our semantic framework.

```
module NotEqual (signal int x1, x2,
            signal bool y) {
    loop {
        if (clk(x1) or clk(x2)
            or clk(y)) {
            emit(clk(x1));
            emit(clk(x2));
            emit(clk(y));
            val(y) = val(x1) != val(x2);
        }
        if (!clk(x1) or !clk(x2)
            or !clk(y)) {
            clk(x1) = false;
            clk(x2) = false;
            clk(y)  = false;
        }
        pause;
    }
}

module Buf (signal int x, y,
         int ?c) {
    int z;

    z  = c;
    loop {
        if (clk(x) | clk(y)) {
            emit(clk(x));
            emit(clk(y));
            next(z) = val(x);
            val(y) = z;
        }
        if (!clk(x) or !clk(y)) {
            clk(x) = false;
            clk(y) = false;
        }
        pause;
    }
}
```

```
module When (signal int  x1,
          signal bool x2,
          signal int  y) {
    loop {
        if (clk(x1) and val(x2)
            and clk(x2) or clk(y)) {
            emit (clk (x1));
            emit (clk (x2));
            emit (val (x2));
            emit (clk (y));
            val (y) = val (x1);
        }
        pause;
    }
}

module Default(signal int x1,x2,y ) {
    loop {
        if (clk(x1) or clk(x2)
            or clk(y) and (!clk(x1)
            or !clk(x2))) {
            emit(clk(y));
            if (!clk(x1)) →
                emit(clk(x2));
            if (!clk(x2)) →
                emit(clk(x1));
            if (clk(x1))
                  val(y) = val(x1)
               else val(y) = val(x2);
        }
        if (!clk(y) | !clk(x1)
            & !clk(x2)) {
            clk(x1) = false;
            clk(x2) = false;
            clk(y)  = false;
        }
        pause;
    }
}
```

Figure 17: Encoding polychronous operators in Quartz

Simulating Signal*'s MoCC in* Quartz. An alternative approach is presented in previous work. We applied this operational framework to define an embedding of Signal into Quartz [42]. This embedding, defined in Figure 17, consists of executing the operational semantics of Signal operators by corresponding Quartz modules. In the figure, a signal x is simply represented by a datatype **signal** t that consists of a boolean clock field **clk**(x) and of a value field **val**(x), of type t. This embedding is of interest on its own, since it allows us to use the existing causality and constructiveness analysis framework and to extend it to checking interpreted Signal programs constructive. It provides a better understanding of the relationship between synchrony and polychrony, between constructiveness and endochrony. It additionally allows us to model the causality problem as a formal verification problem.

10. 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 allows us to better understand the relationship between synchrony and polychrony, between constructiveness and endochrony; and to reason about causality as a formal verification problem. We formulated a constructiveness theory which encompasses the behavior of deterministic synchronous modules and polychronous networks. Along the way, we provided the very first executable operational semantics of Signal.

Acknowledgements
This work is partially supported by INRIA (associate-project Polycore), the Deutsche Forschungsgemeinschaft DFG, by the US Air Force Research Laboratory (grant FA8750-11-1-0042) and the US Air Force Office for Scientific Research (grant FA8655-13-1-3049).

References

[1] G. Berry, The foundations of Esterel, in: G. Plotkin, C. Stirling, M. Tofte (Eds.), Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, 1998, pp. 425–454.

[2] K. Schneider, The synchronous programming language Quartz, Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (December 2009).

[3] N. Halbwachs, A synchronous language at work: the story of Lustre, in: Formal Methods and Models for Codesign (MEMOCODE), IEEE Computer Society, Verona, Italy, 2005, pp. 3–11.

[4] N. Halbwachs, Synchronous programming of reactive systems, Kluwer, 1993.

[5] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone, The synchronous languages twelve years later, Proceedings of the IEEE 91 (1) (2003) 64–83.

[6] B. Titzer, J. Palsberg, Nonintrusive precision instrumentation of microcontroller software, in: Y. Paek, R. Gupta (Eds.), Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM, Chicago, Illinois, USA, 2005, pp. 59–68.

[7] G. Berry, A hardware implementation of pure Esterel, Sadhana 17 (1) (1992) 95–130.

[8] K. Schneider, J. Brandt, T. Schüle, A verified compiler for synchronous programs with local declarations, Electronic Notes in Theoretical Computer Science (ENTCS) 153 (4) (2006) 71–97.

[9] G. Logothetis, K. Schneider, Exact high level WCET analysis of synchronous programs by symbolic state space exploration, in: Design, Automation and Test in Europe (DATE), IEEE Computer Society, Munich, Germany, 2003, pp. 10196–10203.

[10] M. Boldt, C. Traulsen, R. von Hanxleden, Compilation and worst-case reaction time analysis for multithreaded esterel processing, EURASIP Journal on Embedded Systems 2008 (1) (2008) 59–129. doi:10.1155/2008/594129.

[11] N. Halbwachs, F. Maraninchi, On the symbolic analysis of combinational loops in circuits and synchronous programs, in: Euromicro Conference, IEEE Computer Society, Como, Italy, 1995, pp. 42–46.

[12] P. Caspi, M. Pouzet, Synchronous kahn networks, SIGPLAN Notice 31 (6) (1996) 226–238.

[13] G. Kahn, The semantics of a simple language for parallel programming, in: Proceedings of the IFIP Congress, North-Holland, 1974.

[14] P. Le Guernic, T. Gauthier, M. Le Borgne, C. Le Maire, Programming real-time applications with SIGNAL, Proceedings of the IEEE 79 (9) (1991) 1321–1336.

[15] P. Le Guernic, A. Benveniste, Real-time, synchronous, data-flow programming: The language SIGNAL and its mathematical semantics, Research Report 533, Institut National de Recherche en Informatique et en Automatique (INRIA), Rennes, France (June 1986).

[16] O. Maffeis, P. Le Guernic, Distributed implementation of SIGNAL: Scheduling & graph clustering, in: H. Langmaack, W.-P. de Roever, J. Vytopil (Eds.), Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), Vol. 863 of LNCS, Springer, Lübeck, Germany, 1994, pp. 547–566.

[17] L. Besnard, T. Gautier, P. Le Guernic, J.-P. Talpin, Compilation of polychronous data flow equations, in: S. Shukla, J.-P. Talpin (Eds.), Synthesis of Embedded Software – Frameworks and Methodologies for Correctness by Construction, Springer, 2010.

[18] L. Stok, False loops through resource sharing, in: International Conference on Computer-Aided Design (ICCAD), IEEE Computer Society, 1992, pp. 345–348.

[19] M. Riedel, J. Bruck, The synthesis of cyclic combinational circuits, in: Design Automation Conference (DAC), ACM, Anaheim, California, USA, 2003, pp. 163–168.

[20] M. Riedel, Cyclic combinational circuits, Ph.D. thesis, California Institute of Technology, Pasadena, California, USA (2004).

[21] G. Berry, The constructive semantics of pure Esterel, `http://www-sop.inria.fr/meije/esterel` (July 1996).

[22] T. Shiple, G. Berry, H. Touati, Constructive analysis of cyclic circuits, in: European Design Automation Conference (EDAC), IEEE Computer Society, Paris, France, 1996, pp. 328–333.

[23] S. Edwards, Making cyclic circuits acyclic, in: Design Automation Conference (DAC), ACM, Anaheim, California, USA, 2003, pp. 159–162.

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

[25] J. Aguado, M. Mendler, Constructive semantics for instantaneous reactions, Theoretical Computer Science (TCS) 412 (11) (2011) 931–961.

[26] J. Brzozowski, C.-J. Seger, Asynchronous Circuits, Springer, 1995.

[27] G. Berry, The Esterel v5 language primer (July 2000).

[28] M. Mendler, T. Shiple, G. Berry, Constructive boolean circuits and the exactness of timed ternary simulation, Formal Methods in System Design (FMSD) 40 (3) (2012) 283–329.

[29] W. Kautz, The necessity of closed circuit loops in minimal combinational circuits, IEEE Transactions on Computers (T-C) C-19 (2) (1970) 162–166.

[30] R. Rivest, The necessity of feedback in minimal monotone combinational circuits, IEEE Transactions on Computers (T-C) C-26 (6) (1977) 606–607.

[31] W. Howard, The formulas-as-types notion of construction, in: J. Seldin, J. Hindley (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, Academic Press, 1980, pp. 479–490.

[32] S. Malik, Analysis of cycle combinational circuits, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD) 13 (7) (1994) 950–956.

[33] R. Bryant, A switch level model and simulator for MOS digital systems, IEEE Transactions on Computers (T-C) C-33 (2) (1984) 160–177.

[34] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5 (2) (1955) 285–309.

[35] F. Boussinot, SugarCubes implementation of causality, Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis, France (September 1998).

[36] K. Schneider, J. Brandt, T. Schüle, T. Türk, Maximal causality analysis, in: J. Desel, Y. Watanabe (Eds.), Application of Concurrency to System Design (ACSD), IEEE Computer Society, Saint-Malo, France, 2005, pp. 106–115.

[37] K. Schneider, J. Brandt, Performing causality analysis by bounded model checking, in: Application of Concurrency to System Design (ACSD), IEEE Computer Society, Xi'an, China, 2008, pp. 78–87.

[38] E. Sentovich, Quick conservative causality analysis, in: F. Vahid, F. Catthoor (Eds.), International Symposium on System Synthesis (ISSS), ACM, Antwerp, Belgium, 1997, pp. 2–8.

[39] B. Jose, A. Gamatie, J. Ouy, S. Shukla, SMT based false causal loop detection during code synthesis from polychronous specifications, in: S. Singh (Ed.), Formal Methods and Models for Codesign (MEMOCODE), IEEE Computer Society, Cambridge, UK, 2011, pp. 109–118.

[40] M. Nanjundappa, M. Kracht, J. Ouy, S. Shukla, Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework, in: Electronic System Level Synthesis Conference (ESLsyn), IEEE Computer Society, San Francisco, CA, 2012, pp. 24–29.

[41] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, S. Shukla, Constructive polychronous systems, in: S. Artëmov, A. Nerode (Eds.), Logical Foundations of Computer Science (LFCS), Vol. 7734 of LNCS, Springer, San Diego, California, USA, 2013, pp. 335–349.

[42] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, J.-P. Talpin, Embedding polychrony into synchrony, IEEE Transactions on Software Engineering (TSE) 39 (7) (2013) 917–929.

[43] S. Roman, Lattices and Ordered Sets, Springer, 2008.

[44] G. Berry, G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, Science of Computer Programming 19 (2) (1992) 87–152.

[45] K. Schneider, T. Kropf, The C@S system: Combining proof strategies for system verification, in: T. Kropf (Ed.), Formal Hardware Verification – Methods and Systems in Comparison, Vol. 1287 of LNCS, Springer, 1997, pp. 248–329.

[46] J. Brandt, K. Schneider, Separate translation of synchronous programs to guarded actions, Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (March 2011).

[47] SYNCHRON, The common format of synchronous languages – the declarative code DC, Technical report, C2A-SYNCHRON project (1998).

[48] E. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Communications of the ACM (CACM) 18 (8) (1975) 453–457.

[49] K. Chandy, J. Misra, Parallel Program Design, Addison-Wesley, Austin, Texas, USA, 1989.

[50] D. Dill, The Murphi verification system, in: R. Alur, T. Henzinger (Eds.), Computer Aided Verification (CAV), Vol. 1102 of LNCS, Springer, New Brunswick, New Jersey, USA, 1996, pp. 390–393.

[51] H. Järvinen, R. Kurki-Suonio, The DisCo language and temporal logic of actions, Technical Report 11, Tampere University of Technology, Software Systems Laboratory (1990).

[52] A. Gamatié, T. Gautier, P. Le Guernic, J. Talpin, Polychronous design of embedded real-time applications, ACM Transactions on Software Engineering and Methodology (TOSEM) 16 (2).

[53] A. Benveniste, P. Le Guernic, C. Jacquemot, Synchronous programming with events and relations: the SIGNAL language and its semantics, Science of Computer Programming 16 (2) (1991) 103–149.

[54] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann, Polychrony for system design, Journal of Circuits, Systems, and Computers (JCSC) 12 (3) (2003) 261–304.

[55] J.-P. Talpin, D. Potop-Butucaru, J. Ouy, B. Caillaud, From multi-clocked synchronous processes to latency-insensitive modules, in: W. Wolf (Ed.), Embedded Software (EMSOFT), ACM, Jersey City, New Jersey, USA, 2005, pp. 282–285.

[56] A. Benveniste, B. Caillaud, P. Le Guernic, Compositionality in dataflow synchronous languages: Specification and distributed code generation, Information and Computation 163 (1) (2000) 125–171.

[57] D. Potop-Butucaru, B. Caillaud, Correct-by-construction asynchronous implementation of modular synchronous specifications, Fundamenta Informaticae 78 (1) (2007) 131–159.

[58] D. Potop-Butucaru, Y. Sorel, R. de Simone, J.-P. Talpin, Correct-by-construction asynchronous implementation of modular synchronous specifications, Fundamenta Informaticae 108 (1-2) (2011) 91–118.