Polychronous Automata

Paul Le Guernic, Thierry Gautier, Jean-Pierre Talpin, Loïc Besnard

Polychronous Automata

Paul Le Guernic Thierry Gautier Jean-Pierre Talpin
INRIA, Rennes-Bretagne-Atlantique Research Centre, France

Loïc Besnard
CNRS, IRISA, Rennes, France

***Abstract*—This paper investigates the way state diagrams can be best represented in the polychronous model of computation. In this relational model, the basic objects are signals, which are related through data-flow equations. Signals are associated with logical clocks, which provide the capability to describe systems in which components obey to multiple clock rates. We propose a model of finite-state automata, called polychronous automata, which is based on clock relations. A specificity of this model is that an automaton is submitted to clock constraints. This allows one to specify a wide range of control-related configurations, either reactive, or restrictive with respect to their control environment. A semantic model is defined for these polychronous automata, that relies on a Boolean algebra of clocks.**

I. Introduction

The design of embedded systems, and more specifically critical systems, requires the satisfaction of strong, various and heterogeneous constraints such as safety, determinism of embedded programs, threaded or distributed implementation, scheduling in a specific or non specific OS, etc. One way to help designers is to provide them with friendly usable tools supported by strong mathematical semantics. These formal models and methods allow to ensure correctness of components used or defined at each level of the design. The *polychronous model of computation* [1] is such a formal model. Historically related to the synchronous programming paradigm [2] (Esterel [3], Lustre [4]), the polychronous model of computation, implemented in the data-flow language Signal [5], [6] and its environment Polychrony[1], stands apart by the capability to model multi-clocked systems. The synchronous paradigm consists of abstracting the non-functional implementation details of a system and lets one benefit from a focused reasoning on the logics behind the instants at which the system functionalities should be secured. The fundamental notion of *polychrony* consists in the capability to describe systems in which components obey to multiple clock rates. In particular, the Signal language gives the opportunity to seamlessly model embedded systems at multiple levels of abstraction while reasoning within a simple and formally defined mathematical model. A design approach that may be advocated is to allow for a seamless inter-operation of heterogeneous programming viewpoints within the same host model of computation which is the polychronous model. A typical case study from Airbus, for instance, was based on a co-modeling of the doors management system of the A350 [7]. In this case study, functionalities were modeled with synchronous Simulink[2] and a system-level model of the hardware equipment was specified in AADL [8]. The Polychrony toolbox was then used to interpret computations and communications specified in both models in order to synthesize schedulers for sequential and distributed simulation. The experiment was successful, since it has demonstrated that the polychronous model, through its supportive language Signal, may be used as an effective common semantic model for representing or interfacing heterogeneous models. However, Signal is based on a data-flow oriented notation, thus there is sometimes some distance between an actual specification, which may use, for instance, state oriented description, and its semantic encoding as systems of equations. This may cause some practical difficulties, in particular when traceability is a requirement, as is the case in most systems.

In this paper, we investigate the way state diagrams can be best represented in the polychronous model of computation, maintaining the multi-clock characteristic property of the representation. We propose a model of automata, called *polychronous automata*, which is based on clock (or event) relations, and allows one to specify a wide range of control-related configurations, more or less permissive (or, dually, more or less restrictive). A semantic model is defined for these polychronous automata, that relies on the Boolean algebra of clocks, and permits to manipulate these automata, without having necessarily to translate them into data-flow equations.

Related work. Usual automata have been introduced several years ago in data-flow synchronous languages and are used every day in production tools like SCADE [9]. More generally, there have been many attempts to combine heterogeneous programming models. A major problem addressed in Ptolemy is the use of heterogeneous mixtures of models of computation [10]. So-called Modal Models in particular are hierarchical models where the top level model consists of a finite-state machine, the states of which are refined into other models, possibly from different domains [11]. In our approach, heterogeneous designs are expressed in terms of a common semantics, which is that of the polychronous model of computation. In the software system Matlab/Simulink, which is largely accepted in the industry, the Stateflow notation [12] is used to describe modes in event-driven and continuous systems.

Mode-automata [13] were originally proposed to gather advantages of declarative and imperative approaches to synchronous programming and extend the functional data-flow paradigm of Lustre with the capability to model transition systems. Mode-automata have been combined with stream functions in Lucid Synchrone [14]. Related forms of hierarchical state machines include Statecharts [15] and their variants, including UML state machines [16], and SyncCharts [17], associated in particular with Esterel and, more recently, SC-Charts [18], based on an improved definition of (sequential) constructivity [19]. DDFCharts [20], which compose finite-state machines and synchronous data-flow graphs, have multiple clocks; transitions can be driven by different clocks and the instants at which clocks synchronize are seen as a rendez-vous communication.

[1]http://www.irisa.fr/Polychrony
[2]http://www.mathworks.com/products/simulink

Our approach may be distinguished from the others by its capability to model multi-clocked systems and to express clock relations through the automata. A first attempt was made a few years ago to define polychronous mode automata [21], but compared to our current proposal, it did not allow to manipulate automata as specific objects that can be used, for instance, to specify dynamic properties of events. In its spirit, our approach, which is based on constraint specification relating occurrence of events, is relatively close to that taken for Lutin [22], but with a different purpose. In Lutin, statements describe sequences of non deterministic atomic reactions expressing constraints on input/output values. It is used mainly for test sequence specification and generation. In our proposal, constraints relate values and *clocks* of signals.

Although motivated by practical reasons such as the effective combination of heterogeneous programming notations, the main purpose of this paper is not to propose another extension of an existing programming language. Instead, we focus on the definition of a specific model of automata, specially adapted to the polychronous model of computation. Such automata have to relate events by expressing and specifying *clock relations* (or clock *constraints*) between these events. For the definition of polychronous automata, the Signal language is used as syntactic support to express clock equations. Simple examples such as alternating events are used in this paper as they are sufficient to illustrate the basics of the model.

In the next section, we first recall the main operators of the Signal language and their semantics. Then we define in Section III the Boolean control algebra which is used to manipulate clock formulas. In Section IV, we describe the refinement of polychronous programs as automata. In Section V, we highlight different forms of polychronous automata described as equations on signals. Relying on these requirements in terms of expressivity, we define our model of automata in Section VI. Then, regular event expressions for simple forms of automata are proposed in Section VII. Conclusion and future work are drawn in Section VIII.

II. The Signal language

We first introduce the Signal language and its semantics, before to formalize its Boolean control algebra, that is the basis for clock calculus. Signal is a declarative language expressed within the polychronous model of computation. The reader is referred to the bibliography of the Signal language for a detailed description (for instance [23] for an overview, [24], [5] for detailed syntax and semantics).

A Signal *process* defines a set of (partially) synchronized *signals* as the composition of equations. A signal x is a finite ($(\exists n \in \mathbb{N})(x = (x_t)_{t \in \mathbb{N}, t \leq n})$ or infinite ($x = (x_t)_{t \in \mathbb{N}}$) sequence of typed values in the data domain $\mathbb{D}_x$; the indices in the sequence represent logical discrete time *instants*. At each instant t, a *signal* is either *present* and holds a value v in $\mathbb{D}_x$, *absent* and virtually holds an extra value denoted $\perp$, or *completed* and never holds any actual or virtual value for all instants s such that $t \leq s$. The set of instants at which a *signal* x is present is represented by its *clock* $\hat{x}$. Two signals are *synchronous* iff they have the same clock. Clock constraints result from implicit constraints over signals and explicit constraints over clocks.

The semantics of the full language is deduced from the semantics of a core language, and from the Signal definition of the extended features. A Signal process is either an equation $x := f(x_1, \ldots, x_n)$, where f is a function, or the composition $P|Q$ of two processes P and Q, or the binding P/x of the signal variable x to the process P. In this section, we give a sketch of its functional part using data-flow models.

Semantic domains. For a set of values of some type $\mathbb{D}$, we define its extension $\mathbb{D}_\perp = \mathbb{D} \cup \{\perp\}$, where $\perp \notin \mathbb{D}$ denotes the absence of a signal value. The semantics of Signal is defined as least domain fixed point. For a data domain $\mathbb{D}$, we consider a poset $(\mathbb{D}_\perp \cup \{\bullet, \#\}, \leq)$ such that $(\mathbb{D}_\perp, \leq)$ is flat, i.e., $x \leq y \Rightarrow x = y$, for all $x, y \in \mathbb{D}_\perp$ ($\bullet$ and $\#$ denote respectively the presence of a signal and the absence of information). We denote by $\mathbb{D}^\infty$ the set of finite and infinite sequences of "values" in $\mathbb{D}_\perp$. The empty sequence is denoted by ϵ. All n-ary functions $f : \mathbb{D}^{\infty n} \to \mathbb{D}^\infty$ are defined using the convention of noting s a (possibly empty) signal in $\mathbb{D}^\infty$, v a value in $\mathbb{D}$, x a value in $\mathbb{D}_\perp$. As usual, $|s|$ is the length of s, $s_1.s_2$ is the concatenation of s_1 and s_2.

Given a non empty finite set of signal variables A, a function $b : A \to \mathbb{D}^\infty$ that associates a sequence $b(a)$ with each variable of $a \in A$ is named a *behavior* on A. The length $|b|$ of a behavior b on A is the length of the smallest sequence $b(a)$. An *event* on A is a behavior $b : A \to \mathbb{D}_\perp$. For a behavior b on a set of signal variables A, and an integer $i \leq |b|$, $b(i)$ denotes the event e on A such that $e(a) = (b(a))(i)$ for all $a \in A$. An event e on A is said to be empty iff $e(a) = \perp$ for all $a \in A$. The concatenation of signals is extended to tuples of signals. Two behaviors b_1, b_2 are *stretch-equivalent* iff they only differ on non-final empty events (see [1] for more details).

Signal functions. A Signal function is a n-ary (with $n > 0$) function f that is total, strict and continuous over domains [25] (w.r.t. prefix order) and that satisfies:

- *stretching:* $f(\perp.s_1, \ldots, \perp.s_n) = \perp.f(s_1, \ldots, s_n)$
- *termination:* $((\exists i \in 1, n)(s_i = \epsilon)) \Rightarrow f(s_1, \ldots, s_n) = \epsilon$

Stepwise extension. Given $n > 0$ and a n-ary total function $f : \mathbb{D}_1 \times \ldots \times \mathbb{D}_n \to \mathbb{D}_{n+1}$, the *stepwise extension* of f (e.g., $=$, *and*, $+$, etc.) denoted F is the synchronous function that satisfies:

- $F(v_1.s_1, ..., v_n.s_n) = f(v_1, ..., v_n).F(s_1, ..., s_n)$

Delay. Function *delay*: $\mathbb{D} \times \mathbb{D}^\infty \to \mathbb{D}^\infty$ satisfies:

- $delay(v_1, v_2.s) = v_1.delay(v_2, s)$

The infix syntax of $delay(v_1, s)$ is: `s $ init v`$_1$.

Merge. Function *default*: $\mathbb{D}^\infty \times \mathbb{D}^\infty \to \mathbb{D}^\infty$ satisfies:

- $default(v.s_1, x.s_2) = v.default(s_1, s_2)$
- $default(\perp.s_1, x.s_2) = x.default(s_1, s_2)$

The infix syntax of $default(s_1, s_2)$ is: `s`$_1$ `default s`$_2$.

Sampling. Let $\mathbb{B} = \{ff, tt\}$ denote the set of Boolean values. Function *when*: $\mathbb{D}^\infty \times \mathbb{B}^\infty \to \mathbb{D}^\infty$ satisfies:

- for $b \in \{\perp, ff\}$, $when(x.s_1, b.s_2) = \perp.when(s_1, s_2)$
- $when(x.s_1, tt.s_2) = x.when(s_1, s_2)$

The infix syntax of $when(s_1, s_2)$ is: `s`$_1$ `when s`$_2$.

Process. An *equation* is a pair (x, E) denoted `x := E`. An equation `x := E` associates with the variable `x` the sequence resulting from the evaluation of the Signal function f denoted by `E` (defined as a composition of functions). If $A = \{x_1, ..., x_n\}$ ($x \notin A$) is the set of the free variables in `E`, the equation `x := E` denotes a *process* on A, i.e., a set of behaviors on $A \cup \{x\}$; a process is closed by stretchequivalence (thanks to the stretching rule).

The parallel composition of equations defines a process by a network of strict continuous functions connected by signal names. Composition of processes is associative, commutative and idempotent. When it satisfies the Kahn conditions (no cycle, single assignment...), it then is a strict continuous function or Kahn Process Network (KPN) [26], defined as least upper bound satisfying the equations. It further satisfies the termination and stretching properties (it is closed for the stretching relation). It may or may not be synchronous. In the semantics of Signal [1], a process is the set of infinite behaviors accepted by the above "KPN semantics".

A process with feedback or local variables may be not time-deterministic. The semantics of a non deterministic process can be defined using Plotkin's power-domain construction [27]. The input-free equation `x := x $ init 0` is a typical example of not timely deterministic process: `x` holds a sequence of constant value 0 separated by an undetermined number of silent transitions, characterized by an occurrence of $\perp$.

An example of non deterministic process is the equation `x ::= E`, that defines `x` to be equal to `E` when `E` is present and undefined when `E` is $\perp$ (partial definition). This equation is a shortcut for `x := E default x`. A signal `x` can be constructively defined by several equations `x ::= E1`, ..., `x ::= En` in a process, provided that for every pair of equations `x ::= Ei`, `x ::= Ej`, when `Ei` and `Ej` are both present, they hold the same value. If `E1`, ..., `En` do not recursively refer to `x` and if they denote functions, then `(x ::= E1|...|x ::= En)` is a deterministic process.

Partial definitions are very useful in automata where the function that computes the value of a signal often depends on current state. The states being exclusive, the consistency property is satisfied. Partial definitions are used also to define *state variables* the elements of which are present as frequently as necessary. When such a state variable is not explicitly defined, it keeps its previous value.

Derived operators. The following notations (which are derived operators) are used to manipulate clocks, represented as signals of type `event`, always *true* if and only if present.

- *null clock* `^0` (never present)
- *signal clock* `^x`, defined by `x = x` (present, and *true*, when `x` is present)
- *selection* `~b` (a.k.a. `[b]` or `when b`), defined by `^b when b` (present, and *true*, when `b` is present and is *true*)
- *intersection* `x1 ^* x2` by `^x1 when ^x2`
- *union* `x1 ^+ x2` by `^x1 default ^x2`
- *difference* `x1 ^- x2` by `when ((not ^x2) default ^x1)`
- *synchronization* `x1 ^= x2` as `(c := (^x1 = ^x2))/c`

A *synchronized memory* `y := x cell c init x0` is defined by `y := x default (y $ init x0)` and `y ^= x ^+ [c]`. It defines `y` with the most recent value of `x` when `x` is present or `c` is present and *true*. Finally, the Signal term `l1 :: P` associates the label `l1` with the process `P`; a label `l1` is a signal of type `event`. Its clock is the *tick* of the labeled process, `P` (i.e., the upper bound of all the clocks in `P`).

III. Boolean control algebra

We define the syntax and set the axioms of the Boolean control algebra, taking into account *state variables* used to represent states of the automata.

Definition 1. *Considering V a (possibly empty) countable set of signal variables, S a non empty finite set of state variables with $S \cap V = \emptyset$, a* Boolean control algebra *$\Phi(V, S)$ is a tuple $(F_{V,S}, \hat{*}, \hat{+}, \hat{-}, \neg, \mathbf{0}, \mathbf{1}_V)$, where:*

- *$\hat{*}$, $\hat{+}$ designate meet (infimum) and join (supremum);*
- *$\mathbf{0}$, $\mathbf{1}_V$ are the minimum and maximum.*

The set of control Boolean formulas $F_{V,S}$ is the smallest set that satisfies:

- *constants $\mathbf{0}, \mathbf{1}_V \in F_{V,S}$;*
- *atoms $\forall x \in V \cup S$, $\hat{x}, \widetilde{x} \in F_{V,S}$;*
- *unary expressions $\forall f \in F_{V,S}$, $\neg f \in F_{V,S}$;*
- *binary expressions $\forall f, g \in F_{V,S}$, $\hat{+} fg, \hat{*} fg, \hat{-} fg \in F_{V,S}$.*

Parentheses and infix notations can be used in the formulas.

The formula $\widehat{x}$ designates the *clock* of a variable x. The formulas satisfy Boolean axioms: $(F_{V,S}, \hat{*}, \hat{+}, \neg, \mathbf{0}, \mathbf{1}_V)$ is a Boolean algebra. The following supplementary axioms are also considered.

- difference $f \hat{-} g = f \hat{*} \neg g$;
- partition $\forall x \in V \cup S$, $\widehat{x} = \widetilde{x} \hat{+} \neg\widetilde{x}$ and $\widetilde{x} \hat{*} \neg\widetilde{x} = \mathbf{0}$;
- exclusion $\forall s1, s2 \in S$, $\widetilde{s1} \hat{*} \widetilde{s2} = \mathbf{0}$ or $s1 = s2$;
- $\mathbf{1}_\emptyset = \mathbf{0}$.

The clock of an automaton with a non-empty V is defined by $\sum_{x \in V} (\widetilde{x} \hat{+} \neg\widetilde{x}) = \mathbf{1}_V$, and $\forall s \in S$, $\widehat{s} = \mathbf{1}_V$. Formulas in the Boolean control algebra have normal forms: Shannon disjunctive forms (given an arbitrary total order on variables).

Clock hierarchy. In this context, timing analysis mainly refers to analyzing clock relations based on *clock hierarchy*. The clock hierarchy of a process is a component of its *Data Control Graph* (DCG). The DCG is made of a multigraph G and a *clock system* Σ. We refer to [23] for a more complete description. A clock equation is a class of equivalent clock formulas. The clock system Σ is a forest (set of trees) of clock equations: this is why it is called *clock hierarchy*. The clock hierarchy is defined by a relation $\hat{\searrow}$ (*dominates*) on the quotient set of signals by `^=` (x and y are in the same class iff they are synchronous). Informally, a class C dominates a class D if the clock of D is computed as a function of Boolean signals belonging to C and/or to classes recursively dominated by C. A node n of a tree, which is a clock equation, contains also the list of signals $\mathit{signal}(n)$ the clock of which is equal to this class. A tree represents an *endochronous* process: it has a fastest rated clock and the status (presence/absence) of all signals is a *pure flow function* (this status depends neither on communication delays, nor on computing latencies).

The equational nature of the Signal language is a fundamental characteristic that makes it possible to consider the compilation of programs as an endomorphism over Signal programs. We have mentioned a few properties allowing to rewrite programs with rules such as commutativity and associativity of parallel composition. More generally, until the very final steps of code generation (when code generation is an objective), the compilation process may be seen as a sequence of morphisms allowing to rewrite programs as transformed Signal programs. The final steps (C code generation for instance) are simple morphisms over the transformed Signal programs. These transformation steps, to sequential, clustered, or distributed code generation, are described in [23].

IV. Refinement of processes as automata

So far, contrary to what is done for other synchronous languages, including data-flow ones like Lustre (see for instance [28]), no explicit representation of automata was directly produced in the compilation of Signal programs, either for code generation purpose, or as input to formal verification tools. In this section, we propose a method for deriving an automaton from a polychronous program, which relies heavily on the concept of clock.

A given Signal program may be seen as an automaton which contains one single state and one single transition, labeled by a clock. This clock is the upper bound of all the clocks of the program (the *tick* of the program).

The construction of a refined automaton from a Signal program will be based on delayed signals, viewed as state variables (in particular Boolean ones). A state of the automaton is a Signal program with some valuation of its state variables. Transitions are labeled by clocks, which represent the events that fire these transitions. The principle of the construction consists in dividing a given state according to the possible values of a state variable (i.e., *true* and *false* for Boolean state variables, which are considered here) in order to get two states, and thus two new Signal programs. Each one of these two states is obtained using a rewriting of the starting program. Moreover, the absence of value for the state variable (which can be considered as another possible value) is taken into account in the clocks labeling the transitions. The construction of the automaton is a hierarchic process.

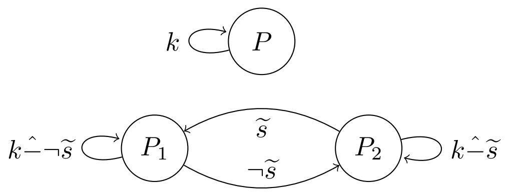

$$\begin{tikzcd} P \arrow[loop left, "k"] \end{tikzcd}$$
$$\begin{tikzcd} P_1 \arrow[loop left, "k \hat{-} \neg\widetilde{s}"] \arrow[r, bend right, "\neg\widetilde{s}"'] & P_2 \arrow[l, bend right, "\widetilde{s}"'] \arrow[loop right, "k \hat{-} \widetilde{s}"] \end{tikzcd}$$

Figure 1. Refinement of state s in P by the automaton $A_1 = (P_1, P_2)$

Figure 1 illustrates the first step of the construction. Initially, the automaton A has one single state, which is the Signal program P, with one transition, labeled by the tick k of P. The construction is started with the valuation of a first state variable, s, in the program P, respectively with *true* and *false*, which gives two new programs, P_1 and P_2. The new programs are obtained by rewriting the previous one, taking into account the considered valuation. This rewriting results generally in simplifications of the programs. The resulting automaton, A_1, now contains two states, P_1 and P_2. The calculus of the transitions consists in computing the clocks of the events that cause a change of state. The transition from P_1 to P_2 occurs when the state variable s, which was *true*, becomes *false*; thus the corresponding clock is $\neg\widetilde{s}$. Conversely, the transition from P_2 to P_1 occurs at the clock $\widetilde{s}$. The transition from P_1 to itself is labeled by the clock k, minus the instants at which there is a transition from P_1 to P_2 (and the same reasoning for P_2). Note that the transitions are not instantaneous. When a clock raising a change of state is present at a given instant, the effective change of state of the automaton takes place at the following instant (with respect to the tick).

The construction of the automaton is an iterative process, by successive valuation of its state variables, s_1, s_2, etc. For instance, the second step would introduce new states, P_{11} and P_{12} from P_1, by discriminating it according to the value of a second variable s_2. One could, equivalently, introduce two others states from P_2. Now, at any refinement step $n > 1$, one could then potentially define 2^n states by iterating the refinement of all sub-processes P_i^{n-1}, of clocks k_i^{n-1} and indices $0 < i \leq 2^{n-1}$ obtained from step $n-1$, by partitioning them according to an nth state variable s_n and by using the same mechanism, Figure 2.

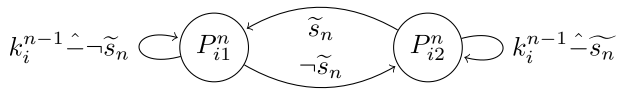

$$\begin{tikzcd} P_{i1}^n \arrow[loop left, "k_i^{n-1} \hat{-} \neg\widetilde{s}_n"] \arrow[r, bend right, "\neg\widetilde{s}_n"'] & P_{i2}^n \arrow[l, bend right, "\widetilde{s}_n"'] \arrow[loop right, "k_i^{n-1} \hat{-} \widetilde{s}_n"] \end{tikzcd}$$

Figure 2. Refinement of state s_n in P_i^{n-1} by the automaton (P_{i1}^n, P_{i2}^n)

This would seem to be an expensive construction, at least in the worst case, in the size of the explicit automaton being an exponential of its number of state variables. Fortunately, however, all Signal programs have a clock hierarchy, defined from the dominance relation introduced in Section III, which is used to represent the control flow of the program in a much more efficient way (and actually optimal, as hierarchies can be normalised and admit a canonical representation). Concretely, most of the 2^n are in practice inaccessible, because of dominance (of a state value by a clock).

Example. Let's for instance consider a Signal program with two state variables s_1 and s_2 such that s_2 is not defined when s_1 is *false*, i.e., s_2 ^= `when` s_1. In other words, s_1 has a higher frequency than s_2, and in the built clock hierarchy, the clock of s_1 dominates s_2. If we construct its automaton as in figure 2, evaluating s_1 first, s_2 second, we would obtain four sub-processes $P^2_{11,12,21,22}$ from P^1_1 and P^1_2. However, partitioning P^1_2 into $P^2_{21,22}$ is useless, since s_2 is not present when s_1 is *false*.

It may further be observed that, when constructing the automaton, the order in which state variables are valuated has an influence on the number of states of the automaton. Our choice is to therefore base this order on the clock hierarchy of the Signal program, using a pre-order depth-first traversal. In this way, more frequent state variables are evaluated before less frequent ones. Note also that when some state variable is valuated, the corresponding program is rewritten, using in particular constant propagation. This results generally in many simplifications, since a number of clocks may become null, thus eliminating corresponding variables.

V. Automaton description in Signal equations

Of particular interest from the previous example is that, in the polychronous framework, the behavior of an automaton may be either reactive, with respect to its environment or context, or restrictive: constrained by clock relations. This can be illustrated on an automaton alternating two events, `a` and `b`. Events `a` and `b` are *constrained* to alternate by the clock relation $a \hat{*} b = 0$, which imposes that they cannot happen simultaneously (the intersection of clocks `a` and `b` is never present). It should further be assumed also that `b` cannot occur in S_1 and `a` cannot occur in S_2. It can be noticed that the occurrences of `a` and `b` are always controlled (or constrained), and control is state dependent.

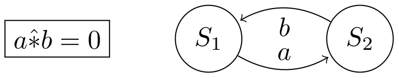


Figure 3. A restrictive behavior

Such an automaton can also be expressed by constraining a and b to occur in either of the automaton states s.

```
s := not (s $ init false) | a ^= [s] | b ^= [not s]
```

A *reactive* behavior, as in Lustre or Esterel, is different. Events a and b are free to occur at any time. An Esterel or a Lustre program does not "control" the delivery of its input signals. A reactive automaton will observe and record the alternating occurrences of a and b, Figure 4, it will not enforce them.

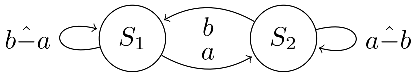


Figure 4. A reactive behavior

In Signal, the observer will be implemented using a couple of equations that monitor alternation using a state variable change and stuttering using another wait.

```
  wait    := change cell (a^+b) init false
| waiting := wait $ init false
| change  :=         (true when a when not waiting)
             default (false when b when waiting)
```

A resettable Esterel program like the famous ABRO is an object which falls in between constrained and reactive behaviors: it emits an output O immediately after receiving both inputs A and B. It is reset when R occurs. So, signal O is control, while others aren't.

```
module ABRO:
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O
each R
end module
```

The automaton for the ABRO is represented Figure 5, where transitions are labeled by Signal clock expressions.

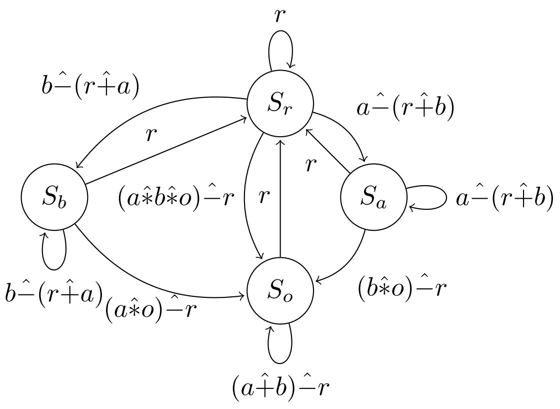


Figure 5. Automaton for ABRO

An equational definition of the ABRO may be specified as follows in Signal, using a state variable for representing the expectation of a and b:

```
  sa := ((false when r) default a) cell b init false
| sb := ((false when r) default b) cell a init false
| wait := r default (false when o) default waiting
| waiting := wait $ init true
| o  := [waiting] when (sa and sb)
```

VI. Explicit structure of automata

Although it is always possible to represent automata by systems of equations, equations are clearly not always the most natural way to represent them. Moreover, in a model-driven engineering context, it is better suited to explicitly represent user-specified and automatically generated automata to maintain high-level semantic properties as well as the traceability of model transformations. We have hence chosen light-weight syntactic extensions to the Signal language in order to introduce explicit representations of automata.

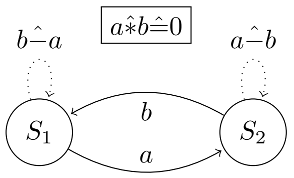
We add a new syntactic category of *process*, called automaton. In such an automaton process, labeled processes represent states, and generic processes such as Transition are used to represent the automaton features. Usual equations can be used in these automaton processes to specify constraints or to define computations. Then comes the question of whether these automata should be only a syntactic structure (in such a way that they would be systematically translated as ordinary equations on signals when compiled), or whether they should be reflected in the polychronous formal model itself. This latter choice has the advantage of allowing a formal manipulation of automata (which may be, or not, translated as equations). For instance, it may be the case that a given behavior is best abstracted as an interface automaton than as a system of clock equations which would require to make explicit some hidden Boolean variables. It is therefore desirable to define a model of "polychronous automata" allowing a possible smooth integration within the polychronous model.

A basic statement for the definition of our automata is that *state change takes time*. It is also assumed in [13], [14], or in SCADE 6. In this way, there is a single state at each logical instant and there is no immediate transition. Such automata should be used to schedule steps, not actions in a step. This drives towards simplicity and is also suitable for high-level mode modeling in an application. In the polychronous framework, transitions will be labeled by clock (or event) expressions named *triggers* and a state is implicitly exited on the upper bound of its triggers. An automaton is *clocked*: it is controllable by an external clock, its *control clock*. There are several possible interpretations of a given automaton: in a permissive view, all non forbidden events are allowed in states; while in a restrictive one, all non allowed events are forbidden in states. By default, we adopt the permissive hypothesis.

A. Notations for constrained automata

To make things concrete, let us write a syntactic representation of the automaton in Fig. 6. This simple automaton has two external events, a and b, and its *control clock* is, implicitly, the upper bound, a ^+ b of the clocks of its inputs. The two states, S1 and S2, are designated by labels, associated here with empty processes. The statement Never (a ^* b) represents the *constraint* of the automaton. This constraint, which much always be respected, can be expressed by a clock formula that is constrained to be null: here, `a ^* b ^= ^0`, expressed as `Never (a ^* b)`. Such constraints can always be expressed as a conjunction of `Never` formulas (which can also be specified as one single `Never` statement with several parameters) or of explicit synchronizations like `Synchro (x, y)` (with `Synchro (x, y)` defined as `Never (x ^- y, y ^- x)`). In this small automaton with two states and two explicit transitions, the initial state is `S1`. We will see below that there are also implicit transitions.

Les us state some vocabulary and notations. For a transition `T = Transition (S1, S2, h)`, `S1` is the source of `T`, `S2` is the target of `T`, `h` is *the trigger of* `T`, denoted *trigger(*`T`*)*, and *a trigger in* `S1`; `h` is a clock expression, that may represent, for instance, a conjunction of events (e.g., `a ^* b` represents the conjunction of events `a` and `b`). The transition `T` is *enabled at* `k` iff `(k ^* h)` is not null and the current state is `S1`.



```
  S1 :: P1()
| S2 :: P2()
| Initial_State (S1)
| Transition (S1, S2, a)
| Transition (S2, S1, b)
| Never (a ^* b)
```

Figure 6. Constrained automaton (with implicit stuttering loops)

A *step* is defined as being an implicit not exiting reflexive (e.g. stuttering) transition. A step `S1--h->>S1` is *enabled at* `k` iff `(k ^* h)` is not null, current state is `S1` and there is no enabled transition. All steps that are not explicitly forbidden are allowed. In the example of Fig. 6, `S1 -- b^-a ->> S1` and `S2 -- a^-b ->> S2` are steps (they appear as dotted transitions). Steps allow for "stuttering" when there is no enabled transition. For a state `S`, exiting on (one of the) enabled transitions is mandatory. A formal model of constrained automata, consistent with the polychronous framework, is proposed in the next section.

B. Formal definition

For an automaton A of signal variables V_A and states S_A, we denote by $F_{A,S}$ the set of *normal form* formulas in the Boolean control algebra $\Phi(V_A, S_A)$ (cf. Section III).

Definition 2. *A* polychronous automaton *A is an epsilon-free automaton defined up to isomorphism (over states) as a tuple $A = (S_A, s_0, R_A, V_A, T_A, C_A)$ where:*

- *S_A is the non empty finite set of states;*
- *s_0 is the initial state;*
- *$R_A \subset S_A \times S_A$ is the transition relation;*
- *V_A is the, possibly empty, finite set of signal variables;*
- *$T_A : (R_A) \to F_{A,S}$ is the function that assigns a formula to a transition;*
- *C_A is the* constraint *of A: it is a formula in $F_{A,S}$ that is (constrained to be)* null *(thus a formula f in $F_{A,S}$ is* null *in A iff $f \hat{*} C_A = f$).*

Remarks. A transition in T_A carries a formula in the Boolean control algebra that represents its trigger. Polychronous automata are subject to clock constraints C_A which are expressed by a formula in the Boolean control algebra. If C_A is $\mathbf{0}$, then the automaton is *constraint-free*; if C_A is $\mathbf{1}_{V_A}$ (i.e., the supremum of the algebra is constrained to be null), all formulas are null. The notation 1_A is used to denote the supremum $\mathbf{1}_{V_A}$.

An automaton with an empty set of transitions is $\mathbb{O}_V = (\{s\}, s, \emptyset, V, \emptyset, \mathbf{1}_V)$, which blocks all occurrences of all variables of V. The automaton with an empty set of variables is $\mathbb{I} = \mathbb{I}_\emptyset = (\{s\}, s, \emptyset, \emptyset, \emptyset, \mathbf{0})$; it is equal to $\mathbb{O}_\emptyset = (\{s\}, s, \emptyset, \emptyset, \emptyset, \mathbf{1}_\emptyset)$.

Example. The automaton in Fig. 6 is defined by $A = (S_A, s_0, R_A, V_A, T_A, C_A)$ with

- $S_A : \{S_1, S_2\}$
- $s_0 : S_1$
- $R_A : \{(S_1, S_2), (S_2, S_1)\}$
- $V_A : \{a, b\}$
- $T_A : (S_1, S_2) \mapsto a,\ (S_2, S_1) \mapsto b$
- $C_A : a \hat{*} b \hat{+} \neg\widetilde{a} \hat{+} \neg\widetilde{b}$

(a, b are events thus $\neg\widetilde{a}$, $\neg\widetilde{b}$ should be null).

A labeled transition is denoted by "$h : s_1 R_A s_2$" meaning that $((s_1, s_2) \in R_A$ and $T_A((s_1, s_2)) = h)$.

Now notions introduced previously can be formalized:

- The *control clock* of an automaton A is 1_A ($= \sum_{x \in V_A} (\widehat{x})$), the supremum of the clocks of its variables.
- In $h : s_1 R_A s_2$, h is *the trigger* of (s_1, s_2) and *a trigger* in s_1.
- The *trigger* of a state s, $\mathit{trigger}_A(s)$, is the upper bound of the triggers of $(s, *)$, where $(s, *)$ stands for all the transitions outgoing from s.

Then it is possible to define the *stuttering clock* of a state as the clock difference between the control clock of the automaton and the trigger of the state (plus the null clock of the state, $C_A(s) = \widetilde{s} \hat{*} C_A$): the *stuttering clock* of a state s is $\tau(s) = 1_A \hat{-} (C_A(s) \hat{+} \mathit{trigger}_A(s))$. Hence the definition of implicit transitions: when the stuttering clock $\tau(s)$ of a state s is not null, there is a silent implicit transition $\tau(s) : s R_A s$ named *step*. Usual properties of automata can be easily extended to polychronous automata:

- A state t is *n-reachable* in A iff s_0 and t are not null and either
  - $n = 0$ and $t = s_0$,
  - $n > 0$ and t is $(n-1)$-reachable in A,
  - $n > 0$ and $(\exists s\ (n-1)\text{-reachable in } A)$ $(\exists h)(h \hat{*} \widetilde{s}$ not null$)(h : s R_A t)$.
- A state t is *reachable* in A iff it is $|S_A|$-reachable in A.
- A state s is *deterministic* if the triggers of its transitions are mutually exclusive: formally, s is deterministic iff $(\forall((s, s_1), (s, s_2)) \in R_A \times R_A)((s_1 = s_2) \vee (T_A((s, s_1)) \hat{*} T_A((s, s_2)) = \mathbf{0}))$.
- An automaton is *deterministic* iff all its reachable states are deterministic.
- A state s is *total* (or *reactive*) iff $\tau(s) \hat{+} (\Sigma_{(s,t)\,\in\,R_A}(\mathit{trigger}_A((s,t)))) = 1_A$.
- An automaton is *total* (or *reactive*) iff all its states are total (we observe that if C_A is not $\mathbf{0}$ then A is not reactive).

C. Polychronous automata algebra

Just like the synchronous composition, the composition (or synchronous product) of polychronous automata corresponds to the conjunction of the behaviors specified by each of them.

Definition 3. *Let $A = (S_A, s_0, R_A, V_A, T_A, C_A)$ and $B = (S_B, t_0, R_B, V_B, T_B, C_B)$ two polychronous automata, their composition is defined by $AB = A|B = (S_{AB}, (s_0, t_0), R_{AB}, V_{AB}, T_{AB}, C_{AB})$, where:*

- *$S_{AB} = S_A \times S_B$,*

- $R_{AB} = \{((s_1,t_1),(s_2,t_2)) \,|\, ((s_1,s_2),(t_1,t_2)) \in R_A \times R_B\}$,
- $V_{AB} = V_A \cup V_B$,
- $(\forall\ st = ((s_1,t_1),(s_2,t_2)) \in R_{AB})\ (T_{AB}(st) = T_{AB}((s_1,t_1)) \hat{*}\, T_{AB}((s_2,t_2)))$,
- $C_{AB} = C_A \hat{+} C_B$.

Note that the constraint of the composed automaton (its null formula C_{AB}) is defined by the clock union of the constraints of the operand automata.

Theorem 1. *The composition of polychronous constrained automata has the following properties:*

- *if A is deterministic, $A|A = A$, it is idempotent;*
- *it is commutative;*
- *it has a neutral element $\mathbb{I} = (\{s\}, s, \emptyset, \emptyset, \emptyset, \mathbf{0})$;*
- *it is associative.*

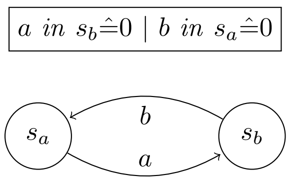
Idempotence for deterministic automata can be proved using induction on n-reachability of states. Associativity can be proved by induction on n-prefix automata (the states of a n-prefix automaton of an automaton A are the n-reachable states of A). Associativity corresponds to context independence and commutativity to order independence.

D. Discussion

The added value provided by the formal model of the polychronous automata is to allow for a smooth integration of automata into the polychronous model of computation of the Signal data-flow language. They have not necessarily to be translated by systems of equations on signals, although such a translation is, of course, possible (comparable translations have been studied in previous works, such as [14] for example, and it is not our purpose here to describe a translation that would not be so different). We have defined a parallel composition (synchronous composition) of polychronous automata. A classical extension of finite automata is also that of *hierarchical automata*, in which states may be non atomic. Here, this can be handled quite simply in the context of the Signal language. It is not detailed in this paper since it does not present new challenges with respect to previous works. Just like labels are syntactically associated with states, labels can also be associated with transitions, and these labels can be used as clocks. The label of a transition is an event signal (a clock), which is *true* (present) when this transition is fired. Actions associated with an automaton can be expressed as polychronous equations (in our case, in Signal), that are composed with the constraints of the automaton. They may use specific events associated with the automaton, such as labels of transitions, but also other typical events such as entering or exiting a given state, etc.

A further remark can be made on permissive versus restrictive interpretation (recall permissive is the default one). The transformation of a given automaton from a permissive interpretation to a restrictive one is obtained as follows by *disabling its steps*. Given a (permissive) automaton A, the high level operation "`strong` A" consists in adding the following constraint for every state s in A: $(1_A \hat{-}\, trigger_A(s)) \hat{*}\, \widetilde{s} = \mathbf{0}$. For an event `h` and a clock `S`, let us write (in a more readable way) "`h in S`" the clock `h ^* [S]`. Consider as example the `A` automaton represented in Fig. 6.

Defining "`automaton alternate = strong A`" adds to `A` the constraints `(a^+b)^-a) in S1 ^= 0` and `(a^+b)^-b) in S2 ^= 0`. Applying constraint reduction, we get the following automaton, represented in Fig. 7.



```
  Sa :: P1()
| Sb :: P2()
| Initial_State (Sa)
| Transition (Sa, Sb, a)
| Transition (Sb, Sa, b)
| Never (b in Sa, a in Sb)
```

Figure 7. "alternate" automaton

VII. Signal Regular Expressions

A further possible extension could be to introduce regular event expressions in order to abstract polychronous automata or represent their *constraint* (or null formula). They can be used also to represent simple automata in which non explicitly allowed events are forbidden.

To define the algebra of regular expressions [29], we consider a Kleene algebra $(A, +, ., *, 0, 1)$, i.e., an idempotent semi-ring $(A, +, ., 0, 1)$, where $(A, +, 0)$ is an idempotent commutative monoid and $(A, ., 1)$ a monoid s.t. $a.0 = 0.a = 0$, $a.(b + c) = a.b + a.c$ and $(a + b).c = a.c + b.c$. A supports a natural, monotonic, partial order $(a \le b)$ iff $(a + b = b)$ and star definition satisfying $1 + a.a* \le a*$, $1 + a * .a \le a*$, $b + a.x \le x \Rightarrow a * .b \le x$ and $b + x.a \le x \Rightarrow b.a* \le x$. Our objective is to represent events and event formulas as regular expressions (extended) with counting. We compare with the related property specification language PSL [30].

Signal regular event expressions are defined on a (finite) vocabulary made of the equivalence classes of event formulas. Values h are event formula classes (in place of $\{h\}$ in PSL) and neither the empty set 0 nor $1 = \{\epsilon\}$ have PSL representation. Both 0 and 1 should remain implicit, as part of the event algebra, with no explicit syntax.

Operators are concatenation, often denoted $S_1.S_2$, $S_1;S_2$ here, as in PSL; union $S_1 + S_2$ ($S_1|S_2$ in PSL); and star $S*$. Usual notations may be utilized for positive $S+ = S;S*$ and option $S? = 1{+}S$. Usual reduction rules apply. Other operators are defined as well, such as fusion $S : T$ (same as PSL) and synchronous product $S|T$. Finally, regular expressions with counting are considered [31], where counters of the form $S[n]$ are inductively defined by $S[0] = 1$ and $S[m+1] = S;S[m]$.

As an example, the constraint of the alternating automaton represented in Fig. 3, $C = (a \hat{*}\, b)$, can equivalently be expressed as the regular event expression `((a ^- b) + (b ^- a))*`. The alternating automaton could itself be alternatively expressed by the composition of two regular event expressions consisting of the negation of the constraint `(a ^* b)*` and of its transitions `(a;b)*`, which yields `((a ^- b);(b ^- a))*`.

VIII. Conclusion

We have presented a model of finite-state automata, called polychronous automata, that integrates smoothly with data-flow equations in the polychronous model of computation. They define transition systems to express explicit reactions together with properties, in the form of Boolean formulas over logical time, to constrain their behavior. The implementation of such automata amounts to composing explicit transition systems with a controller synthesized from the specified constraints. Polychronous automata are being integrated in the open-source version of the Polychrony toolset. They may be used to specify behaviors (and constraints) and to *abstract* behaviors, as the result of a formal calculus. A special class of automata is that which may be represented by regular event expressions, for which a specific formal calculus could be further developed. Such regular expressions would be used as a powerful mechanism to express dynamic properties of polychronous processes. Finally, this formal model of automata supports the recommendations adopted by the SAE committee on the AADL to implement a timed and synchronous behavioural annex for the standard [32].

References

[1] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *Journal of Circuits, Systems and Computers*, vol. 12, no. 03, Jun. 2003, http://hal.inria.fr/docs/00/07/18/71/PDF/RR-4715.pdf.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages twelve years later," *Proceedings of the IEEE, Special issue on Modeling and Design of Embedded Systems*, vol. 91, no. 1, 2003, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.1117.

[3] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Sci. of Computer Program.*, vol. 19, no. 2, pp. 87–152, 1992, http://dx.doi.org/10.1016/0167-6423(92)90005-V.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[5] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with Signal," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991, http://hal.inria.fr/inria-00540460.

[6] A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2009, http://www.springer.com/engineering/circuits+%26+systems/book/978-1-4419-0940-4.

[7] H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P. Le Guernic, A. Toom, and O. Laurent, "System-level co-simulation of integrated avionics using Polychrony," in *ACM Symp. on Applied Computing*, TaiChung, Taiwan, Mar. 2011, http://hal.inria.fr/inria-00536907/en/.

[8] "Aerospace Standard AS5506A: Architecture Analysis and Design Language (AADL)," 2009.

[9] G. Berry, "Scade: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007.

[10] S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee, "A modular formal semantics for Ptolemy," *Math. Structures in Computer Science*, vol. 23, pp. 834–881, 2013, http://chess.eecs.berkeley.edu/pubs/877.html.

[11] E. A. Lee and S. Tripakis, "Modal models in Ptolemy," in *Proceedings of 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT 2010)*, October 2010, pp. 1–11, http://chess.eecs.berkeley.edu/pubs/700.html.

[12] G. Hamon and J. Rushby, "An operational semantics for Stateflow," in *Fundamental Approaches to Software Engineering: 7th International Conference (FASE)*, ser. LNCS 2984. Springer, 2004, pp. 229–243.

[13] F. Maraninchi and Y. Rémond, "Mode-automata: a new domain-specific construct for the development of safe critical systems," *Science of Computer Programming*, vol. 46, no. 3, pp. 219–254, 2003, http://dx.doi.org/10.1016/S0167-6423(02)00093-X.

[14] J.-L. Colaço, B. Pagano, and M. Pouzet, "A conservative extension of synchronous data-flow with state machines," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. ACM, 2005, pp. 173–182, http://doi.acm.org/10.1145/1086228.1086261.

[15] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987, http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[16] Y. Wang, J.-P. Talpin, A. Benveniste, and P. Le Guernic, "A semantics of UML state-machines using synchronous pre-order transition systems," in *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ser. ISORC '00. IEEE Computer Society, 2000, pp. 96–103, http://dl.acm.org/citation.cfm?id=850984.855510.

[17] C. André, "Semantics of SyncCharts," I3S Laboratory, Sophia-Antipolis, France, Tech. Rep. ISRN I3S/RR–2003–24–FR, April 2003.

[18] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially constructive statecharts for safety-critical applications," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2014.

[19] R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, and O. O'Brien, "Sequentially constructive concurrency—A conservative extension of the synchronous model of computation," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013.

[20] I. Radojevic, Z. Salcic, and P. Roop, "Design of distributed heterogeneous embedded systems in DDFCharts," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 2, pp. 296–308, 2011, http://dx.doi.org/10.1109/TPDS.2010.69.

[21] J.-P. Talpin, C. Brunette, T. Gautier, and A. Gamatié, "Polychronous mode automata," in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, ser. EMSOFT '06. ACM, 2006, pp. 83–92, http://doi.acm.org/10.1145/1176887.1176900.

[22] P. Raymond, Y. Roux, and E. Jahier, "Lutin: a language for specifying and executing reactive scenarios," *EURASIP Journal on Embedded Systems*, 2008.

[23] L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin, "Compilation of polychronous data flow equations," in *Synthesis of Embedded Software*. Springer, 2010, http://hal.inria.fr/inria-00540493.

[24] L. Besnard, T. Gautier, and P. Le Guernic, "SIGNAL V4-INRIA version: Reference Manual," 2010, http://www.irisa.fr/espresso/Polychrony/documentation.php.

[25] S. Abramsky and A. Jung, "Domain theory," in *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1994, vol. 3, pp. 1–168.

[26] G. Kahn, "The semantics of a simple language for parallel programming," *Proceedings of the IFIP Congress 74, Stockholm, Sweden*, pp. 471–475, 1974.

[27] G. Plotkin, "A powerdomain construction," *SIAM Journal on Computing*, vol. 5, pp. 452–487, 1976.

[28] N. Halbwachs, P. Raymond, and C. Ratel, "Generating efficient code from data-flow programs," in *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[29] D. Kozen, "A completeness theorem for kleene algebras and the algebra of regular events," in *Logic in Computer Science*, 1991, pp. 214–225.

[30] "IEEE standard for property specification language (PSL)," *IEEE Std 1850-2005*, pp. 1–143, 2005.

[31] W. Gelade, M. Gyssens, and W. Martens, "Regular expressions with counting: Weak versus strong determinism," *SIAM Journal of Computing*, vol. 41, no. 1, pp. 160–190, 2012.

[32] L. Besnard, E. Borde, P. Dissaux, T. Gautier, P. Le Guernic, and J.-P. Talpin, "Logically timed specifications in the AADL: a synchronous model of computation and communication (recommendations to the SAE committee on AADL)," Technical Report RT-0446, Apr. 2014. [Online]. Available: https://hal.inria.fr/hal-00970244