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/espresso/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 remind 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 $\mathbb{D}_t$, we consider the poset $(\mathbb{D}_\perp \cup \{\bullet, \perp\}, \leq)$ and that $(\mathbb{D}_\perp, \leq)$ is flat, i.e., $x \leq y \Rightarrow x = y$, for all $x, y \in \mathbb{D}_\perp$. 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) signals 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 $f : \mathbb{D}^n \to \mathbb{D}$, 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*: $\mathbb{D}_i^\infty \times \mathbb{B}^\infty \to \mathbb{D}_i^\infty$ 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 stretch-equivalence (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 $\perp$. This equation is a shortcut for `x := E default x`. A signal `x` can be constructivey 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`
- *signal clock* `^x`, defined by `x = x`;
- *selection* `~b` (a.k.a. `[b]` or `when b`) by `^b when b`
- *intersection* `x1^*x2` by `bx1 when bx2`
- *union* `x1^+x2` by `bx1 default bx2`
- *difference* `x1^-x2` by `when (not bx2) default bx1`
- *synchronization* `x1^=x2` as `(c := (bx1 = bx2))/c`.

A *synchronized memory* `y := x cell c init x0` is defined by `y := x default (y $ init x0)` and `y b= x b+ [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 `l :: P` associates the label `l` with the process `P`; a label `l` 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 $\forall x \in V \cup S$, $\hat{x}, \mathbf{x} \in F_{V,S}$;*
- *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 $\mathbf{b}$ 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 $\forall x \in V \cup S$, $\mathbf{b} = \mathbf{x} \hat{+} \neg\mathbf{x}$ and $\mathbf{x} \hat{*} \neg\mathbf{x} = \mathbf{0}$
- exclusion $\forall s1, s2 \in S$, $\mathbf{s1} \hat{*} \mathbf{s2} = \mathbf{0}$ or $s1 = s2$.
- $\mathbf{1}_\emptyset = \mathbf{0}$;

The clock of an automaton with a non-empty V is defined by $\hat{\sum}_{x \in V}(\mathbf{x} \hat{+} \neg\mathbf{x}) = \mathbf{1}_V$, and $\forall s \in S$, $\mathbf{b} = \mathbf{1}_V$. 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 $signal(n)$ 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 for instance (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. R

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.

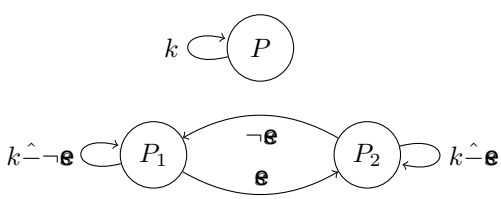

Figure 1. Refinement of state s in P by the automaton $A_1 = (P_1, P_2)$

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 $\neg\mathbf{e}$. Conversely, the transition from P_2 to P_1 occurs at the clock $\mathbf{e}$. 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.

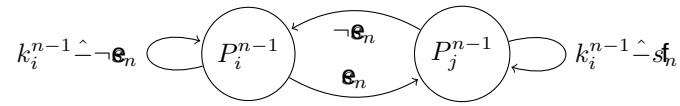

Figure 2. Refinement of state t in P_1 by the automaton (P_{11}, P_{12})

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 true, i.e., s_2 `b= 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 **??**, evaluating s_1 first, s_2 second, we would obtain four sub-processes $P_{11,12,21,22}$ from $P_{1,2}$. However, partitioning P_2 into $P_{21,22}$ 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 $a\hat{*}b\hat{=}0$, which imposes that they cannot happen simultaneously. It should further be assumed also that `b` cannot occur in S_1 and `a` cannot occur in S_2. It can be noticed the occurrences of `a` and `b` are always controlled (or constrained), and control is state dependent.

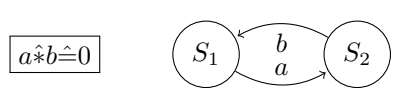

Figure 3. A restrictive behavior

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.

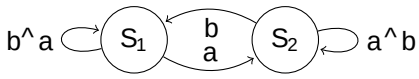

Figure 4. A reactive behavior

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.

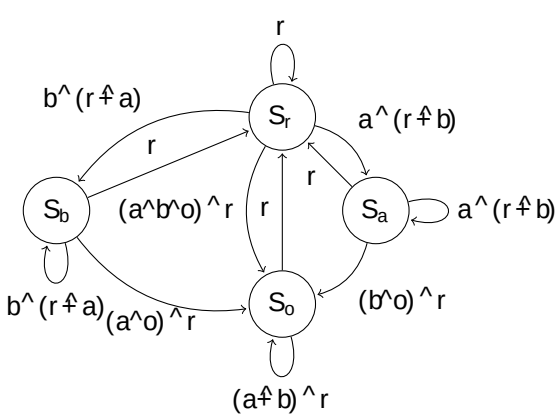

Figure 5. Automaton for ABRO

An equational de nition of the ABRO may be speci ed 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-speci ed 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 de ne 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 re ected 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 de ne a model of "polychronous automata" allowing a possible smooth integration within the polychronous model.

A basic statement for the de nition 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 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 speci ed as one single Never statement with several parameters) or of explicit synchronizations like Synchro (x, y) (with Synchro (x, y) de ned 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.